Introduction to Jackson Design Method: JSP and a little JSD

Nicholas Ourusoff

Table of Contents

Introduction

There are two reasons for writing this book. First, I believe that JSDOOP--coupling the object-based modeling of JSD with object-oriented implementation--is a promising method for information system development. Second, I believe the clear and seminal thinking of Michael Jackson about program and information system design methodology deserves more attention than it has received in the United States; and his contributions have often been misrepresented.[1]

1. The promise of JSDOOP

In the Spring of 1991, I experimented with implementing a JSD specification into an object-oriented programming (OOP) language (Smalltalk). JSD seemed to me then to be object-based in the sense that it begins with a model of the real world in terms of a set of entities (objects), their actions (behavior), and constraints. In fact, as I found later, JSD is object-based in the most fundamental sense--the structure of a program or information system is based on the structure of the problem. This is the single most basic principle throughout Michael Jackson's writing. As John Cameron puts it:

> "Jackson System Development (JSD) and Object-Oriented Design (OOD) have one major--arguably central--principle in common; namely that the key to software quality lies in the structuring of the solution to a problem in such a way as to reflect the structure of the problem itself. There should be a simple and demonstrable correspondence between a (real world) component of the problem and a (software) component of the solution. The two methods also use similar concepts to describe the problem domain (or 'real world'). It is considered to consist of identifiable objects ('entities' in JSD) and operations that are either performed or suffered by these objects ('actions' in JSD}."[2]

My investigations showed that indeed a JSD entity mapped into a Smalltalk object, with JSD actions mapping into methods and an entity's state vector into an object's instance variables.

There is overwhelming evidence that JSD specifications can be directly implemented in OOP even without proving the concept by implementing JSD specifications:

(1) When we examine what an entity is in JSD, we find that it is consistent with an OOP object.

A JSD entity has the following properties:

---

[1] As an example, in one of the articles surveying OOD in CACM, Oct 1991, JSD is said to be an acronym for "Jackson Structured Design". In fact, JSD stands for "Jackson System Development". More importantly, anyone familiar with Jackson's methodology knows that Jackson argues forcefully against structured design methodology which is associated with functional decomposition.

[2] "JSD and Object Oriented Design" by J. R. Cameron and A. Birchenough in [Ca89], p. 305.

(a) entities of the same type form a class; the program text for all individual entities is the same;

(b) an entity has different states, corresponding to different actions it performs or suffers over time; the state vector of an entity consists of all of its local variables and a text pointer to its process text;

(c) associated with each action of an entity is process text that models that action in the real world;

(d) each entity may also have connected to it additional functions

(e) entities in the real world communicate with entity process models by messages (serial data stream) that transmit real-world information about actions that an entity performs or suffers

In total, all of these properties are consistent with the objects in OOP languages. So, an entity should map easily into an OOP object.

(2) JSD specifications are in principle executable, that is, the program text can be constructed from an entity's structure and the structure of functions superimposed on the initial network of entities using JSP. Implementation is by program transformation, of which there are three main techniques:

(a) writing process texts in a form which allows them to be easily suspended and reactivated so that their execution can be scheduled explicitly;

(b) separating process state-vectors from their process text, so that only one copy of the process text need be kept of each type of process, while as many copies of state vectors are kept as there are instances of the entity;

(c) breaking process texts into pieces which can be more conveniently loaded and executed in a conventional environment

But all three techniques can be readily implemented simply by using an OOP language as follows:

(a) In an OOP, entities are activated whenever a message is sent to them, inactive otherwise. They remember their state, and this can certainly include their text pointer. Thus, I see no need for the program inversion, the transformation of a program into a variable state (resumable) subroutines.

(b) Any instance of an OOP class inherits the methods of that class; in other words, the program text of instances is stored once as part of the object class, not with each instance.

(c) Finally, OOP methods are precisely the dismemberment of the process text into convenient modules--we typically have a method for each action and for each function associated with an entity.

Thus, although JSD has its own implementation methods, object-oriented programming (OOP) languages have features that make it very tempting to discard JSD's implementation methods in favor of a direct mapping into an OOP language.

(3) OOP objects communicate with messages

In JSD, the connection between an entity in the real world and an entity in an information system is usually by serial data stream connection, in which the real world entity produces a message for each action performed (or suffered).

In OOD, objects communicate by messages (a serial data stream). Since state vectors are part of an object, state vectors are inspected by sending messages as well.

In summary, JSD specifications are object-based. The implementation of JSD specifications are more naturally implemented with an OOP language, which contain the essential features needed to implement a JSD network of communicating entity processes.

2. Jackson's Contributions to Design Methodology

Michael A. Jackson has made original contributions to program and information systems design methodology. He originated the program design methodology known as Jackson Structured Programming (JSP)--his book, Principles of Program Design (1975) has been rightfully called a classic. Building on the ideas of JSP, he developed together with John Cameron and co-workers, the Jackson System Development (JSD) method for designing information systems.

Jackson's thinking about program and information systems design was often at odds with prevailing opinion. In the early 1970's, Jackson advised against flowcharts as a program design tool and invented Jackson structure diagrams. With Dan McCracken he early articulated dissatisfaction with the traditional life cycle concept, arguing that it was stultifying, and presented iterative prototyping as an alternative. He sharply criticized top-down, functional decomposition, arguing that we should first deal with what the real-world is about, and only later deal with what a system is supposed to do. Whereas data modeling results in a static model, Jackson argued that information systems should model the real world dynamically: JSD models the actions of entities--their real-time behavior. He pointed out that stepwise refinement doesn't provide a method. In contrast, his constructive method provides checks at each step on the correctness of design. Finally, he suggested that formal proofs of information systems are unlikely to be convincing because of their length, and that establishing correctness of a specification--a specification that can be directly executed after suitable correctness-preserving transformations--is a more promising approach to software validation.

Jackson is at his pedagogical best reasoning qualitatively about programs and systems. He instructs us through examples. He shows us recurring design dilemmas,

and teaches us how to resolve them. We are persuaded of each of his discoveries about program design. His unifying insight that programs and systems both model problem domains led him to extend the main ideas of JSP from the domain of programs to that of systems.

In the first phase of JSD, the developer specifies a model, based on discussions with the user, that reflects the structure of the problem domain in terms of the behavior of real-world entities. The specification leads directly--seamlessly--to an entity's program text. JSD is object-based.

Jackson is a 'structurist'. He takes a static "bird's eye" view of a program or entity process, arguing against the error-prone flow-of-control mentality that asks "What happens next?", asking instead "What is the underlying structure of the problem?" New control structures and compiler methods are needed to preserve the structural integrity of programs that require backtracking and inversion. If software must be optimized for performance purposes, this should be done only after the design reflecting the problem structure has been completed.

JSP and JSD are grounded in the simple qualitative notions of serial data stream, sequential process, and regular expression. Jackson's ideas have a strong affinity to those of C. A. Hoare, and JSD has been shown to be theoretically consistent with Hoare's investigations of parallel processing.

Jackson's work has had a seminal influence on the research of others (e.g., Cameron, Sanden, Zave) and on program and information systems design pedagogy in Europe and elsewhere.

Jackson's original investigations of information systems lead us to ask, 'Are not information systems--models of communicating sequential processes--fundamental objects of study in computer science?'

Preface

This text evolved from a long-standing interest in Jackson methodology that began when I learned, as a programmer-analyst at the World Health Organization, to use JSP to specify programs that were subsequently coded by other programmers. Later, I practiced JSP as a programmer involved in validating population census data for Senegal and Guiné-Bisau. I first taught JSP as part of a course in program design (1979), and later gave a full course in JSP (1986). I taught elementary JSD as part of courses in systems analysis (1985), in management information systems (1988-89), and in introductory computing (1991). During the Fall of 1991 while a visiting lecturer at Petrozavodsk State University (Russia), I prepared a set of lectures on JSP and JSD in a course on programming systems that I team-taught with Dr. Anatoly V. Voronin of the Department of Applied Mathematics and Cybernetics. I gave a workshop in Jackson methodology at the ACM SIGCSE Technical Symposium in March, 1992.

JSP is teachable. Many programmers have expressed their experience that JSP gave them insights about program design that they never had before--for the first time, they understood how to design programs that they had been writing without really understanding them. JSP can and should be taught in any computer information systems (CIS) curriculum as part of the first and second courses in programming. JSP is language-independent, and can be taught in any introductory programming course. JSD should be taught after JSP, as a first course in information systems design or software engineering.

A data model can be derived from a JSD specification. Data is associated with actions (events) of an entity process; the data constitutes the state vector of the entity process. Although data models, such as the relational model, can be taught independently of systems development, database design is properly understood, not as a starting point for modeling an information system, but rather as part of the implementation phase of system development.

In the Spring of 1991 I explored object-oriented design with some students. I had the intuition that JSD entities and actions were closely related to object-oriented objects and methods, and I used this seminar to implement a JSD specification in Smalltalk.

JSD is an object-based method of analysis. A JSD specification can be seamlessly implemented using an object-oriented programming language: entities, actions, and attributes of a JSD specification map into objects, methods and instance variables of an OOP language. I have termed the method of implementing a JSD specification using an OOP language "JSDOOP".

0 Introduction

In the chapters that follow, we will explore how to design programs and develop information systems. The approach we use is the software development methodology of Michael Jackson. In fact, the methodology is one, but is known by two acronyms: JSP, Jackson Structured Programming, a method for designing programs; and JSD, Jackson System Development, a method for designing information systems. JSD evolved from--may be viewed as a superset of--JSP.

JSD is object-based, that is, JSD models the behavior of the objects of interest in a user's problem domain. A JSD specification is not some set of abstractions that only programmers and analysts can understand, but consists of the same objects that the users of the system know from their day-to-day activities. This is the essence of what object-based means. The third part of these lectures deals with coupling a JSD specification with object-oriented implementation--what I have termed JSDOOP. We present examples mapping a JSD specification into an implementation using the object-oriented programming language Smalltalk.

Michael Jackson invented JSP from 1972-74 and JSD with John Cameron and other colleagues who worked for Michael Jackson Systems, LTD during the late 1980s. Jackson has written two books, "Principles of Program design" in 1975, which describes JSP, and "System Development" in 1983 which describes JSD. Another primary source material for JSP and JSD is "JSP & JSD: The Jackson Approach to Software Development" by John Cameron (1983 and 1989). Other sources on JSP and JSD may be found in the bibliography.

Jackson's writing excels in clarity and expository approach. If you are interested in understanding Jackson methodology, you cannot do better than to read his own works.

We will start with JSP first.

1 Program Structure

1.1 Introductory Remarks

At the outset, it is worth taking note of several themes in Jackson's thinking:

(1) Design is about structure, about the relation of parts to the whole. The basic function of program flow charts--especially during the 1960's and early 1970's--was to show the flow of control in a program. A flow chart examines the dynamic representation of a program: "What happens next?" In Jackson's thinking, the design of a program follows from the static structure of a program's text: "What is the relationship of parts to the whole?" Since program design is concerned with program structure rather than with program execution, we shouldn't use flow-charts as a tool for designing programs.

(2) There is a difference between getting a program to work and getting it right. A program may work, but may be wrong--it may be difficult to read, may not model the problem to be solved, may be difficult to maintain.

If a program is right, e.g. it has the correct structure, it will be easier to read and maintain. And, a program coded from a correct design is quicker to test, since there will be fewer logic errors requiring redesign.

(3) JSP is a constructive design method. By the phrase "constructive design method", we mean that steps are defined and guidelines given at each step to check the correctness of the design so far.

In the 1960's, we used modular programming as a design method. But what are the criteria for deciding on what becomes a module? Unfortunately, there is no decision procedure to guide the designer in the choice of modules.

Likewise, In the 1970's, step-wise refinement was proposed as a design method by E. Dijkstra. Design proceeded by top-down decomposition using the control structures of structured programming (sequence, selection and iteration). But how do we decompose a problem top-down? There is no decision procedure to guide the designer. Moreover, the biggest decomposition decision must be made right away--the first decomposition--when we have little experience with the problem at hand. Finally, we may question whether stepwise refinement constitutes a method, since it doesn't provide decision criteria to guide the designer during each step of the design.

Like step-wise refinement, JSP, as the "SP" in "JSP" indicates, is a structured programming methodology, e.g. it relies heavily on control structures for sequence, selection and iteration; however, it is not a top-down, but rather a constructive method in which there are criteria that guide the designer at each step of the design process. In JSP we *construct* a model of the task to be solved in the form of a data structure. This data model guides the design of the program.

(4) Jackson gives us a rule about optimizing programs for efficiency. The rule is as follows:

Don't optimize!
If you have to, do it as the last step, after you have designed the program properly.

The reasons for the rules about optimization are (1) optimization is often unnecessary, and (2) optimization distorts the structural correspondence between a program and the problem it models. Thus, optimization tends to obscure the meaning of a program, making it more difficult and expensive to maintain.

1.2 An example: Printing a multiplication table[3]

A multiplication table is to be generated and printed. The required output is:

```
1
2       4
3       6       9
4       8       12      16
...     ...     ...     ...
10      20      30      40      50      60      70      80      90      100
```

The table is to be printed on a line printer using only the basic statements for writing lines of text.

Here is a badly designed program to solve the problem:

```
program mult_table (input, output);
    var
        row_no, col_no, k: integer;
        line: array[1..10] of integer;
    procedure displayline;
        const
            blanks = '    ';
        var
            col_no: integer;
    begin
        write(' ');
        for col_no := 1 to row_no do
            if line[col_no] = 0 then
                write(blanks)
            else
                write(line[col_no] : 4);
        writeln
    end;        {displayline}
    procedure computeline;
        procedure computelement;
        begin
            col_no := col_no + 1;
            line[col_no] := row_no * col_no
        end;        {computelement}
    begin
        displayline;
        row_no := row_no + 1;
        col_no := 0;
```

[3]This example is adapted from Jackson, M. A. [1], pp. 2-7.

```
        while row_no <> col_no do
            computelement
    end;        {computeline}
begin
    writeln;
    row_no := 1;
    line[1] := 1;
    for k := 2 to 10 do
        computeline;
    displayline
end.
```

The program design is based on a flow-chart. It works correctly, producing the required output. The coding itself conforms to the tenets of structured programming: **while** statements control iteration, and there are no **go to** statements. But the structure is hideously wrong.

Consider what would be required to revise the program to produce any of the following outputs:

(i) print the upper-right triangular half of the table instead of the lower-left triangular half; that is, print:

```
1   2   3   4   5   6   7   8   9   10
    4   6   8   10  12  14  16  18  20
        9   12  15  18  21  24  27  30
                        ...
                        81  90
                                    100
```

(ii) print the lower-left triangular half of the table, but upside down; that is, with the multiples of 10 on the first line and 1 on the last line;

(iii) print the right-hand continuation of the complete table; that is, print:

```
 11   12   13   14   15   16   17   18   19   20
 22   24   26   ...  ...  ...  ...  ...  ...  ...
 ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
110  120  130  140  150  160  170  180  190  200
```

Each of these changes should be easy to make. The first change affects only the choice of which numbers are printed within each line; instead of printing line[1] up through line[row_no], we wish to print line[row_no] through line[10]. We should be able to make local changes to the program--perhaps in the third and fourth lines of **computeline**--but we cannot. Instead, we essentially need to rewrite the entire program! We are similarly defeated by the second and third changes.

The essence of the difficulty is this: we are given simple and local changes to output specifications: in the first case, to alter the choice of numbers on the line; in the second case, to later the order of printing the lines; in the last case, to alter the choice and values of numbers to be printed in each line. We therefore look to make corresponding local changes to the program. But where is the program component that determines the choice of numbers to be printed? Where is the component that determines the order of the lines? Where is the component that determines the values of the numbers? The answers are not so simple as we had hoped.

Superficially, **computeline** appears to process each line. In fact, however, **computeline** prints line[row_no] and generates line[row_no+1]. So, **computeline** is executed 9 times, and the 10th line is printed in the main program. In short, the program does not model the structure of the problem.

The program should have been as follows:

```
program mult_table (input, output);
    var
        row: integer;
        line: array[1..10] of integer;
    procedure clearline;
        var
            col_no: integer;
    begin
        for col_no := 1 to 10 do
            line[col_no] := 0;
    end;                {clearline}
    procedure displayline;
        const
            blanks = '    ';
        var
            col_no: integer;
    begin
        write(' ');
        for col_no := 1 to 10 do
            if line[col_no] = 0 then
                write(blanks)
            else
                write(line[col_no] : 4);
        writeln
    end;            {displayline}
    procedure computeline;   {compute a line}
        var
            col_no integer;
        procedure computelement;
        begin
            line[col_no] := row_no * col_no;
        end;            {computelement}
    begin
        for col_no:= 1 to row_no do
            computelement;
    end;            {computeline}
begin
    for row_no:= 1 to 10 do
        begin
            clearline;
            computeline;
            displayline
        end;
end.
```

The program processes the whole table; the procedure **computeline** processes each line; the procedure **computelement** processes each number; the table consists of 10 lines and **computeline** is executed 10 times. Each line consists of row_no numbers, and **computeline** executes **computelement** row_no times. There is a perfect correspondence between program structure and problem structure.

We can produce (i) - (iii) by the simple and local program changes shown below:

(i) in **computeline**: **for** col_no := row_no **to** 10 **do**
(ii) in **computeline**: **for** col_no := 10 **downto** (11-row_no) **do**
in **computelement**: line[col_no] := (11-row_no)*col_no
(iii) in **computeline**: **for** col_no := (10+col_no) **to** 20 **do**
in **computelement**: line[col_no] := row_no*(10+col_no)

The program was designed using a structure diagram.

1.3 Structure Diagrams, Program Structure and Data Structure

Clearly, the example of the multiplication table problem shows that a badly designed program can be costly and difficult to maintain. A compelling reason for constructing well-designed programs is to minimize maintenance costs. The key is to produce programs whose structure corresponds to the problem it solves.

One lesson to be learned is that one should not use flow-charts as a design tool: design is about structure, and flow-charts, as the name suggests, is about flow-of-control. When using flow charts as a design tool, the programmer, instead of thinking about the structure of the program, will think about its execution in the computer.

A more positive lesson is that program structures should be based on data structures. There are deep reasons why this is so, and they are depicted in the following section.

Exercises:
(a) Make modifications to the badly-designed program to produce each of the outputs given.
(b) Make modifications to the well-designed program to produce each of the outputs given.

2 Jackson Structure Diagrams

2.1 Jackson Structure Diagrams

Design is about structure, about the relation of parts to the whole. Programs consist of the following parts or components:

(i) elementary components

Elementary components have no parts. Examples are elementary statements in a programming language or primitive operations of a machine.

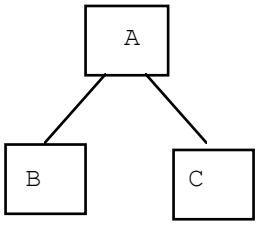
Sometimes, using bottom-up design, we will extend a programming language with new operations. For example, if we need to manipulate matrices, we can define an abstract data type, matrix, together with arithmetic operations. We could then multiply two matrices, for example, with a statement such as matmult(a, b), where a is an m by n array and b is an n by p array.

(ii) composite components

There are three types of composite components--components having one or more parts:

(a) sequence

A sequence is a composite component that has two or more parts occurring once each, in order. In the Jackson structure diagram shown on the left below, A is a sequence consisting of parts B and C. B occurs once, followed by C. To the right of the structure diagram is a textual representation, known as Jackson structure text, of the structure diagram. Pseudocode representation of the structure diagram is shown at the far right.

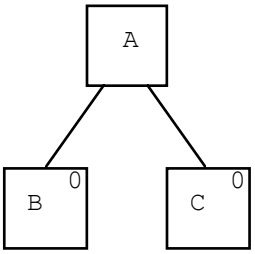
Jackson structure diagram	**Jackson structure text**	**Pseudocode**
	**A seq** do B; do C; **A end**	**begin** do B; do C; **end**

(b) selection

A selection is a composite component that consists of two or more parts, only one of which is selected, once. In the structure diagram below, A is a selection consisting of

parts B and C. Either B or C is selected, not both. Jackson structure text and pseudocode representations of the structure diagram are shown to the right.

Jackson structure diagram



Jackson structure text

```
A sel    <cond-1>
  do B;
A alt    <cond-2>
  do C;
A end
```

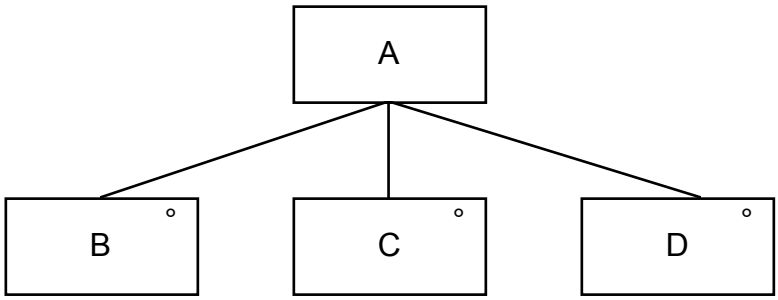
Pseudocode

```
if  <cond-1>  then
  do B;
else if   <cond-2>  then
  do C;
endif
```

In the structure text, the condition for selecting component B or C is written explicitly to the right of the selection header. Note that the condition must be evaluated before we can determine which component we have.

Whereas in most programming languages, condition-1 would be evaluated before condition-2 in the example above, no such ordering is implied by the structure diagram. Consequently, condition-2 cannot in the structure diagram be expressed under the assumption that condition-1 has been evaluated and is not true; rather, condition-2 must explicitly express the condition for which component B is selected without reference to the condition governing the selection of component A.

Note also that the normal interpretation of the **if then...else if...then...endif** statement allows for a null action if neither condition is met. However, the structure diagram indicates that either B or C must be selected. To allow for the case when neither of the conditions for B or C is met, we would draw the structure diagram shown below:

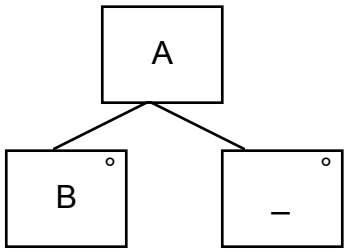


The condition for D would be **not** (cond-1 **or** cond-2). To express null action, the component D would do nothing.

From this example, we see that structure diagrams are a general design tool that can and should be explicit in depicting the structure of program components.

Sometimes we have a situation in which data occurs or doesn't, depending on some condition. This form of degenerate selection is depicted by the following structure diagram and text representation:

Jackson structure diagram



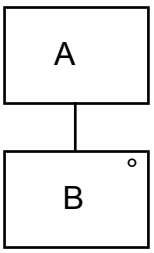
Jackson structure text

```
A   sel    <cond>
      do B;
A   end
```

Pseudocode

```
if    <cond> then
    do B;
endif
```

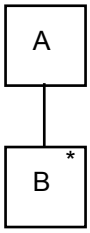
The structure diagram can be abbreviated as shown below:



(iv) iteration

An iteration is a composite component that consists of one part that repeats zero or more times. In the diagram below, A is an iteration containing a part B which repeats 0 or more times. The Jackson structure text and pseudocode representations of the structure diagram are shown to the right of the structure diagram..

Jackson structure diagram



Jackson structure text

```
A iter      <cond>
  do B;
A end
```

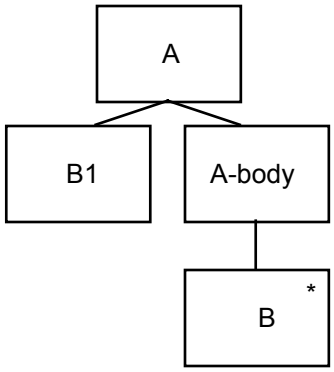
Pseudocode

```
while       <cond>
    do B;
endwhile
```

The **while...endwhile** construct rather than the **repeat...until** form of indefinite iteration will always be used for two reasons. First, this form, with the condition test at the

beginning, is the most general, including the case of an iteration with a part that iterates zero times. The **repeat...until** form has no condition on entry, and its component part is always executed once. The condition for subsequent repetitions indicates that there is something different between the context of the first and subsequent occurrences.

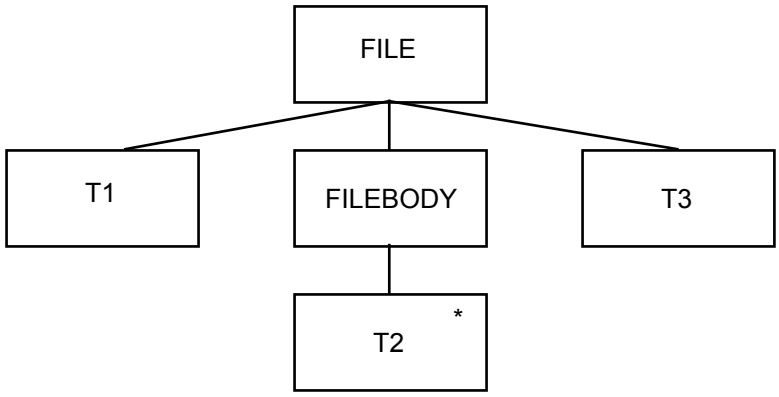
Sometimes, if an iteration must have at least one occurrence of the iterated part, we will show this explicitly as shown below:



Usually, the first occurrence has a different context--as when the first occurrence requires special processing--and it is thus proper to depict the iteration in this explicit form.

Consider the structure diagram below, depicting a file that contains three record types, T1, T2 or T3, containing the values 1, 2, or 3 respectively in the field, **rectype**.[4]

Jackson structure diagram



Jackson structure text

```
PFILE   seq
          do T1;
        PBODY   iter
                  do T2;
        PBODY   end
          do T3;
PFILE   end
```

The structure text corresponding to the structure diagram is shown to its right.

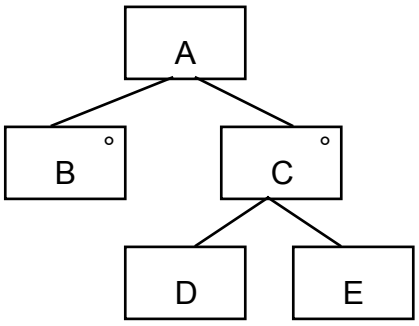
The question that arises is, 'What is the correct condition to write for the iteration, PBODY?" It seems simple to write, "until T3", but this would be a mistake. PBODY

[4] adapted from Jackson, 1975, p. 26

iterates a component, T2, so the explicit condition for the iteration is "while T2", e.g. "while **rectype** = 2. If we use the condition "until T3", we are relying on a property of the specification of FILE not FILEBODY. If the specification of FILE changed, so that a T4 record is interspersed between FILEBODY and T3, we would have to modify the condition controlling the iteration of PBODY, even though the specification of the component FILEBODY has not changed. An accumulation of small changes of this kind can have a large effect on the cost of program maintenance. The guiding principle is to code explicitly the conditions that specify the processing of each program component.

Generally, we will be explicit in depicting composite structures. If a sequence is part of a component that is not the root of the structure, however, we will relax the explicit **begin....end** demarcation for sequence, as is shown in the following example:

Jackson structure diagram



Jackson structure text

```
A sel    <cond-1>
  do B;
A alt    <cond-2>
  do D;
  do E;
A end
```

Pseudocode

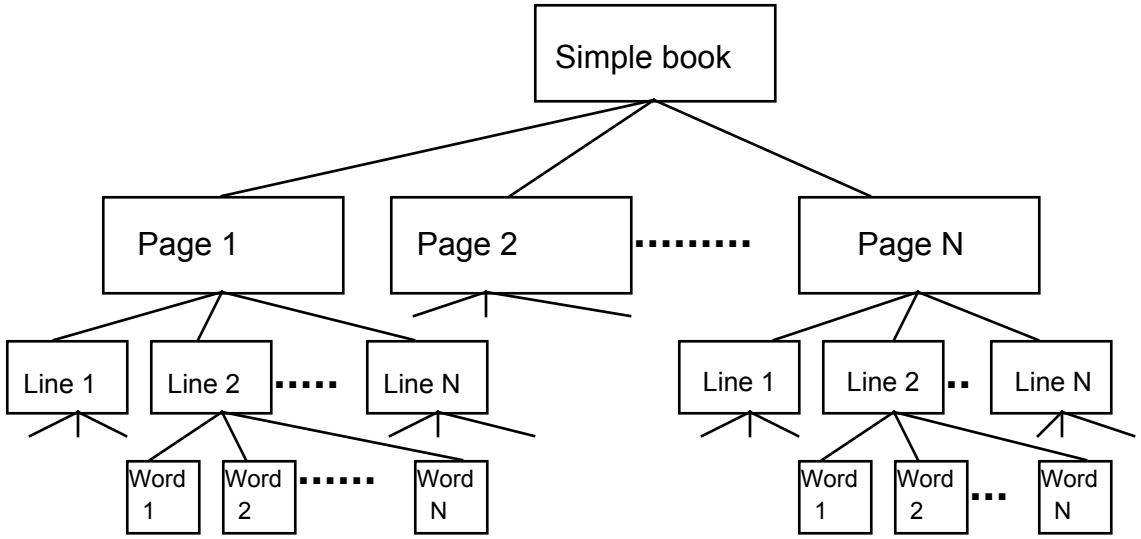
```
if  <cond-1>  then
  do B;
else if   <cond-2>  then
  do D;
  do E;
endif
```

The sequence C, consisting of components D and E, is not explicitly demarcated in the structure text or pseudocode.

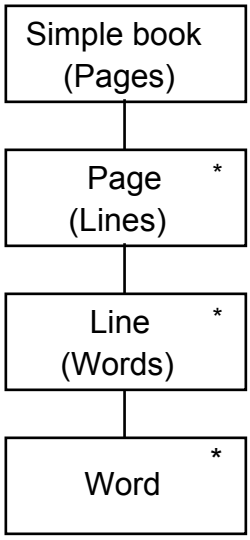
2.2 Examples

(i) A simple book consists of pages; a page consists of lines of text; a line consists of words.

Here is a first attempt at a structure diagram:



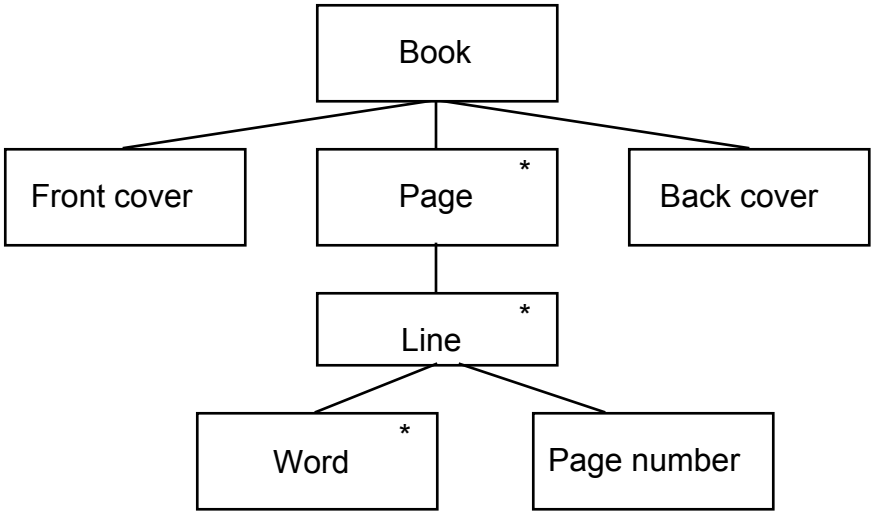
We can simplify this effort replacing each sequence by an iteration:



We see from this example that iteration is a generalization of sequence. Note that a simple book consists of pages; a simple book (pages) is an iteration--the part that iterates is a single page. Similarly, a page is equivalent to lines; a page (lines) is an iteration--the part that iterates is a single line. And so on with line (words).

(ii) A book consists of a front and back cover with pages in between. Each page consists of lines of text; each line consists of words. At the bottom of each page is a page number.

A first effort to draw the structure diagram for a book as described above might yield something like the following:

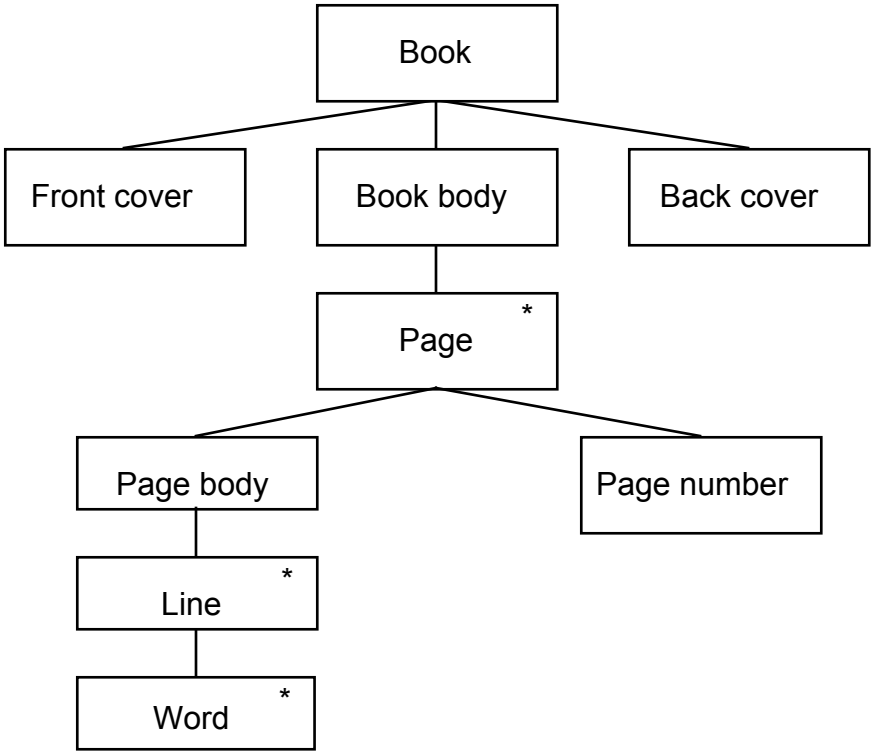


This structure diagram is incorrect, first of all, formally, that is grammatically: For, we may ask, what kind of component is a book? It appears that a book is an iteration, since there is a part, page, that iterates. On the other hand, book appears to be a sequence, since there are three consecutive parts--front cover, page, and back cover But this is an impossible situation--a composite component must either be a sequence, a selection or an iteration--it cannot be a hybrid combination. While a sequence does have three parts, none of them repeats--each occurs exactly once; While an iteration has a part that repeats, it has one and only one part.

A similar, formal error in the diagram occurs in the part that shows a line as having two parts, one of which iterates. So, what kind of component is a line? It cannot be an iteration, since it has two parts; it cannot be a sequence, since one part iterates. It is an impossible construct that violates the grammar of structure diagram construction.

A third error is in the placement of page number. From the diagram, a page number appears after all of the words in each line rather than after all lines have occurred.

The correct structure diagram for a book is as shown below:



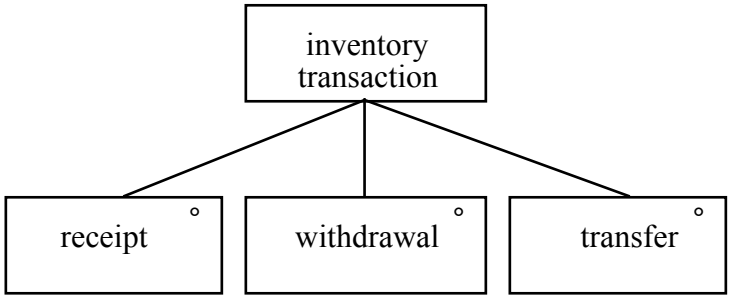
In the structure diagram, note that we have created a name for an iteration that is part of a sequence: "Book body" is the part of a sequence that comes after the front cover but before the back cover; "book cover" is an iteration of page. Likewise, "page body" is the part of a page that comes before the page number.

In general, we have to create a name for any composite component that is part of another component to satisfy the formal rules of structure diagram construction..

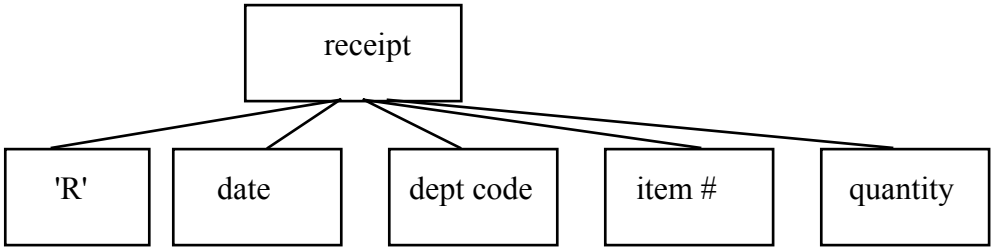
(iii) An inventory transaction for a warehouse

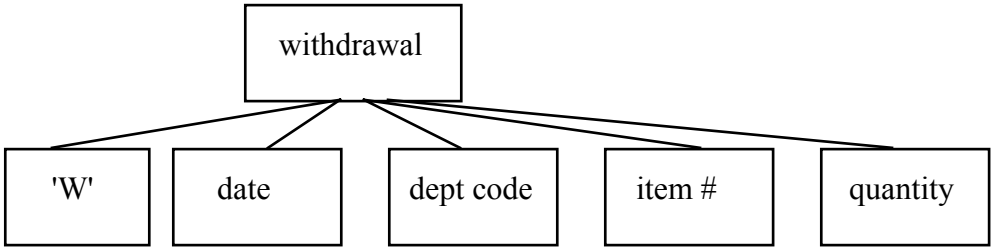
Three types of transaction are defined: a receipt of inventory, indicated by a code of "R"; a withdrawal of inventory, indicated by a code of "W"; and a transfer of inventory, indicated by a code of "T". In the case of a receipt, the data included on the transaction is date of receipt, department code, item number, and quantity received; in the case of a withdrawal, the data included is date, department code, item number, and quantity withdrawn; for a transfer, the data included is transaction date, department code issuing the inventory, item number and quantity issued, and department code to which the inventory is being transferred.

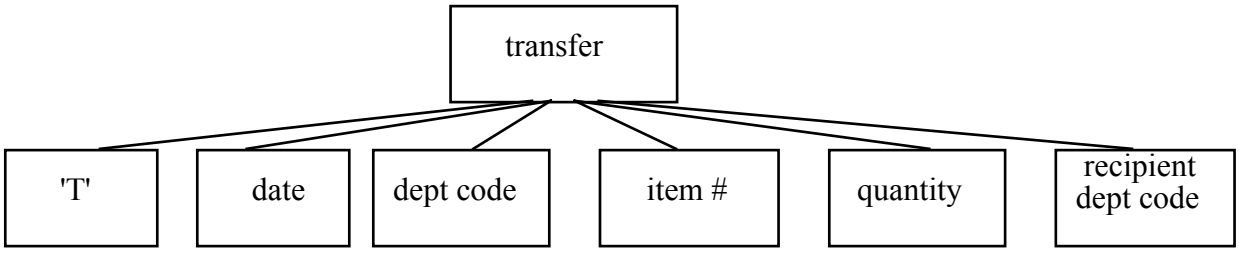
The data structure for the inventory transaction described above is shown in the Jackson structure diagram below:



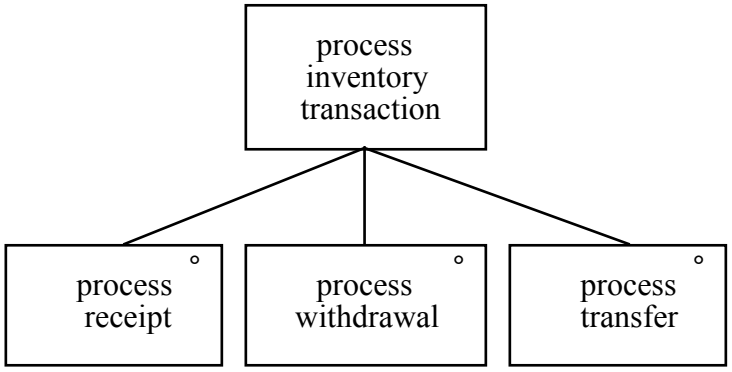
The data structures for each transaction type are shown below:







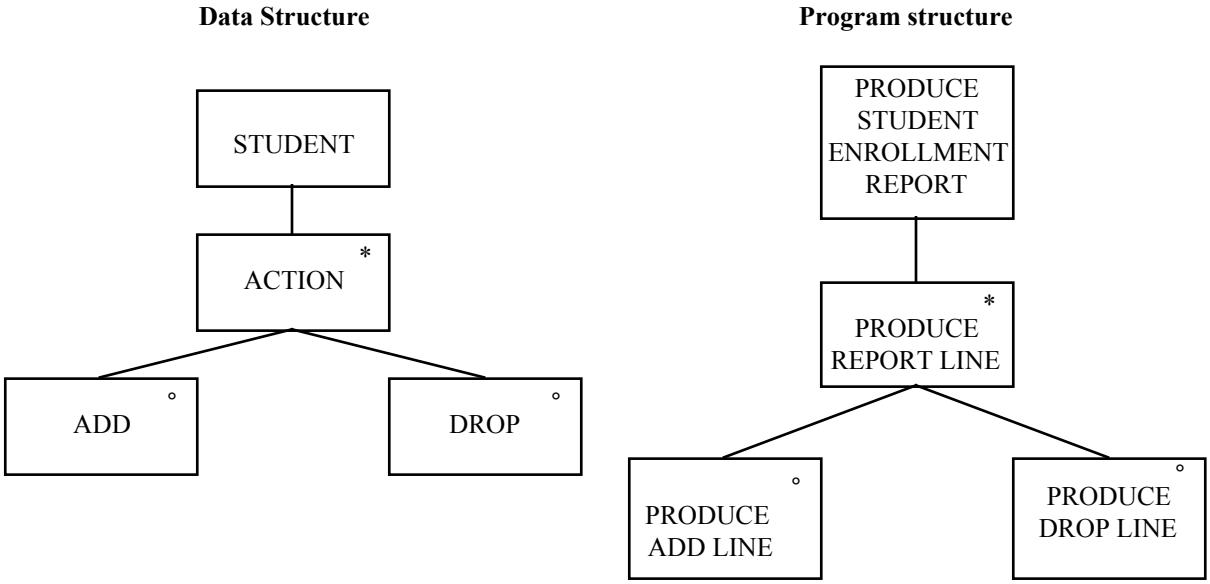
Suppose we wish to process inventory transactions. The gross structure of the program component to process a transaction is evidently:



Note that the structure of the program component to process a customer transaction is identical to its data structure. We simply use a verb in each node of the diagram to express the action to process the data.

(iv) Student registration in courses

Students in a university add or drop courses. The student provides a code, 'A' for add or 'D' for DROP; his or her identification code; and the course code. We are required to produce an enrollment activity log for each student. The structure of a student's actions is shown below in the data model at left.

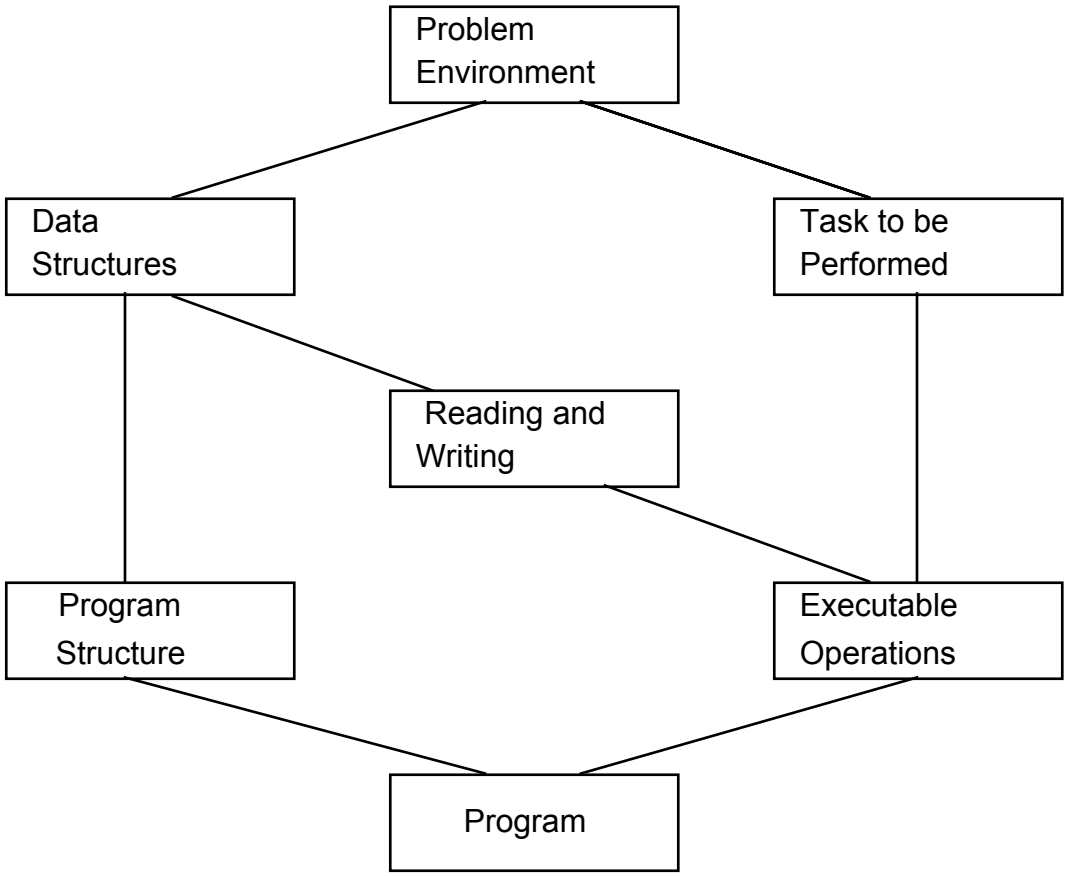


If we consider a program to display a student's enrollment activity, the program structure is evidently that shown to the right of the data model structure diagram. Note the correspondence between the program structure and the data structure. For a student, the program produces a report; for each student action, the program has a component to write a report line for that action; for an add action, the program has a component to produce an add line, while for a drop action, there is a component to produce a drop line.

2.3 Program structure based on data structure

More fundamentally, examples (iii) and (iv) in the previous section illustrate the basis of Jackson methodology: We **model** a problem first, using a data structure (model) to capture the problem structure. The program structure is derived from the data model. Thus, **program structure reflects problem structure**. The situation is shown in the diagram below:[5]

Program structure based on data st ructure.



The problem environment is that part of the real world that a computer system models. In the case of student registration, the real world consists of students who add or drop courses. Of course, there are constraints on a student's behavior: he cannot add a course that doesn't exist; he can't drop a course he hasn't previously added; he can't add a course he is already enrolled in.

The computer system sees the world through the data structures that model a student's possible actions. A serial stream of a student's actions over time--a student file-

[5] Jackson, 1975, p. 10

-contains a code of 'A' or 'D', the student's identification number, and the course identification number occur in each record. The file is a model of the student's actions. Each record models an action of the student.

The program consists of a series of operations to be executed by the computer. Some of these are associated with moving around the data structures: we must read the next action record for a student and write the next report line. Other actions are more directly associated with the tasks to be performed. For example, we may need to keep track of how many actions a student made. Each time a student adds or drops a course, there must be an operation "count := count + 1"; and the variable count must be properly initialized at the start and printed out at the end.

For both types of operation, we can associate the operation with a component of the data structures on which the program is based.

JSP is based on these design ideas. We begin by modeling the problem and expressing the model in the form of one or more data structures. From the data structures, we form a program structure. We consider the tasks to be performed, and list the operations needed. Then we allocate the operations to the appropriate component of the program structure.

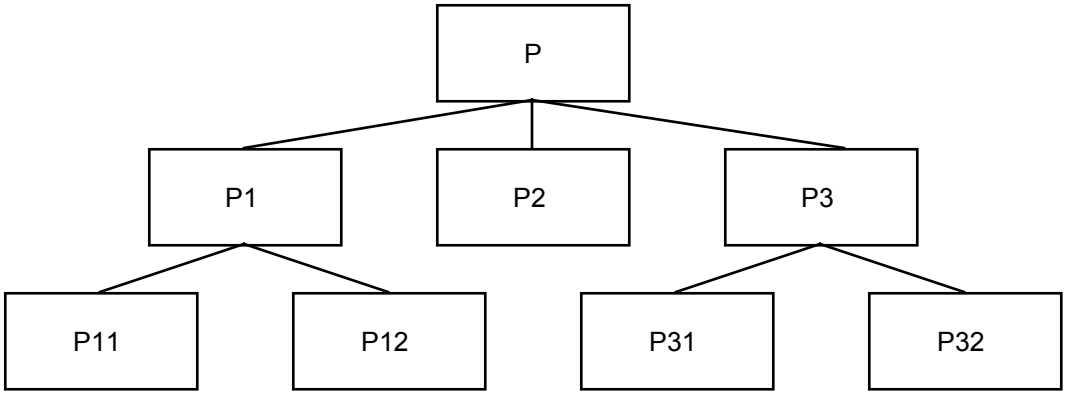
2.4 Elementary versus generalized components

The machine we use provides us with a set of elementary data types and a set of elementary operations. In examining a problem, we may decide that we need data types and operations at a more abstract level in order to solve the problem. In effect, using bottom-up design, we modify our initial machine, M' and create a new machine, M" that has new elementary data types and operations. For example, suppose we are programming in PASCAL. Our programming environment has already modified our hardware to create what we may call a PASCAL-machine. When we operate on a Boolean data type, we are not concerned with its machine representation, nor with the machine instructions to execute an assignment statement such as

```
b := true;
```

where b is a Boolean variable. Now, suppose we wish to add a matrix data type, together with operations for doing matrix arithmetic. We may extend our programming language to include a data type, matrix, together with operations matadd, matsub, matmult and matdiv, together with matrix constants, matzero and matident to represent the zero and identity matrices. In effect, we have extended our elementary operations by bottom-up design.

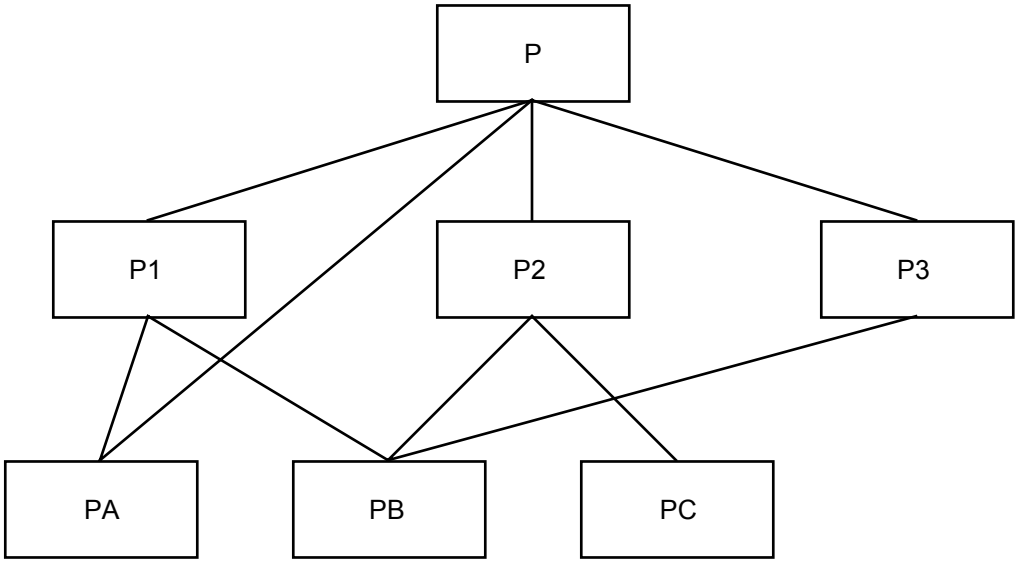
On the other hand, when we design a program with JSP, constructing the program structure from a data model of the problem, we build a tree structure such as the following:



In a tree structure, each component depends on one and only one component higher up in the hierarchy.
Here again there are no generalized components. A change to P2 will have no effect on any other component because each component has structural integrity.

In top-down design, the attention span of the designer is limited to a less complex problem than the original one. Thus, problem P is dissected into subproblems P1, P2 and P3 and each of these is similarly dissected, resulting as before in a tree structure such as that given above. We will see later that JSP is not a top-down method, but rather a constructive method of design. But like top-down design, JSP creates a program structure in which each component preserves structural integrity.

Contrast this with the following picture that uses generalized components:



We may wonder how the design process was accomplished. PC depends only on P2, but PA depends on both P1 and P, while PB depends on P1, P2 and P3. What part of component PB will need to be changed if component P1 is changed? Will this change affect the way PB works as a component of P3?

Clearly, the design process was not top-down. In some sense we have optimized since this design has only seven components, whereas our original had eight. Our components are generalized. We may have saved valuable storage space. But in the process, we have lost design integrity, and increased the burden of future maintenance.

There are two lessons here: First, we can design new, more general elementary operations using bottom-up design to transform our programming environment. Second, created generalized components is an optimization technique; as will be discussed later, optimization should only be attempted after a correct design has been derived.

Exercises

(i) Compose a single structure diagram that depicts the warehouse inventory transaction and each transaction type described in section 2.2 above.

(ii) (a) Write a Pascal record structure for the warehouse transaction described in section 2.2 above

(b) Write a Pascal block of code to process a warehouse transaction described in section 2.2 above.

(iii) Draw a structure diagram for each of the following Pascal declarations:
(a) var a : array[1..10] of integer;
(b) var b : array[1..10] of array[1..20] of real;
(c) var c : array[1..10] of array[1..20] of packed array[1..30] of char;

(iv) Draw a structure diagram for the file, employees, declared below:

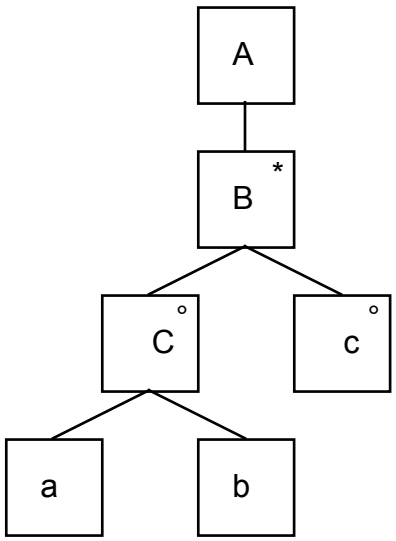
```
const
    maxraises = 50;
    maxchildren = 25;
type
    alpha = packed array[1..20] of char;
    date = record
        month : (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);
        day : 1..31;
        year : integer
        end;
    sex = (male, female);
var
    employees : file of record
            lastname, firstname : alpha;
            ssn : integer;
            birthdate : date;
            maritalstatus : (single, married, divorced, widowed);
```

```
        numberofraises : 1..maxraises;
        salaryhistory : array[1..maxraises] of record
                begindate : date;
                salary : integer;
                jobtitle : alpha;
            end;
        numberofchildren : 1..maxchildren;
        child : array[1..maxchildren] of record
                birthdate : date;
                firstname : alpha;
            end;
        case s : sex of
            male : ();
            female : (maidenname : alpha);
    end;
```

(v) For each of the following regular expressions below, interpret the regular expression as a data structure and draw a structure diagram.

(a)	((a*|b*)*c)|d	(b)	(a*)*|b|cd)
(c)	((a|b|cd)*)*	(d)	a|(b(cd)*)
(e)	ab(c|d*)*	(f)	a*b*(c|d)*
(g)	((ab)*)*|cd	(h)	(a|bc|d*)*
(i)	(a*b|cd*)*	(j)	((a*)*(b|c)*)d
(k)	(a|b)*c(d*)*	(l)	a(b|(cd*))*
(m)	ab(c|d*)*	(n)	a*|(b*c*)|d
(o)	(a*|b|c*)*d		

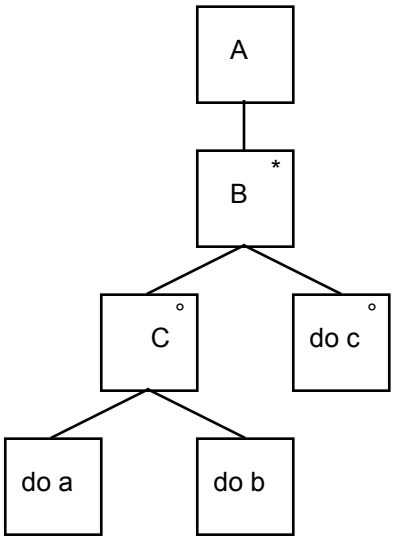
Example: (ab|c)*



(vi) For each of the regular expressions in (v), interpret the regular expression as a program. Draw the corresponding structure diagram. and give the equivalent Jackson structure text and pseudocode.

Example: (ab|c)*

Structure Diagram



Jackson structure text

```
A   iter
  B   select
    C   seq
          do a;
          do b;
    C   end
  B   alt

        do c;
  B   end
A   end
```

Pseudocode:

```
dowhile <cond1>
  if <cond2>
    begin
          do a;
          do b;
    end
  else

        do c;
  endif
endwhile
```

(vii) For each of the regular expressions in (vi), write specifications for a program whose structure corresponds to the regular expression.

Example: (ab|c)*

"For lunch you may have either soup and crackers or a salad. You may have as many servings of either as you wish."

3. JSP: Basic Design Method and the Single Read-ahead rule

3.1 Basic Design Method

The basic design method in JSP consists of the following steps:

1 Draw a system diagram
2 Draw a data structure for each input and output file
3 Draw a single data structure based on correspondences between the input and output data structures; this data structure forms the basic program structure
4 List the operations needed by the program, For each, ask "Where does it belong (in what program part?)" "How many times does it occur?" Allocate the operations to the basic program structure.
5 Translate the program structure into text, specifying the conditions for iteration and selection.

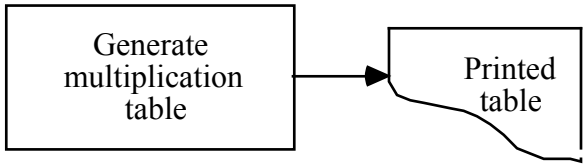
To illustrate the basic design technique, let's begin with a simple example--the multiplication table that we considered earlier. A multiplication table is to be generated and printed. The required output is:

```
1
2       4
3       6       9
4       8       12      16
...     ...     ...     ...
10      20      30      40      50      60      70      80      90      100
```

The steps of the basic JSP design method for this example are:

1 Draw system diagram.

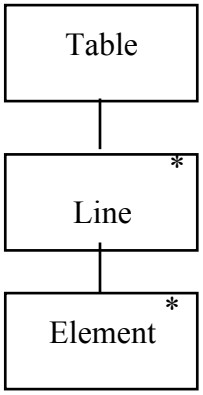
In a system diagram we record what the inputs to and outputs from the program are. In our example, we have no input. The system diagram shows the program to generate multiplication table producing the printed table as output:



In simple problems such as this one, we can omit showing this step explicitly, since the system diagram is more or less self-evident.

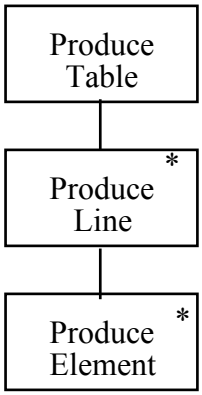
2 Draw data structures

We record our understanding of the problem environment by modeling it with suitable data structures. Our understanding of the multiplication table is expressed in the data structure diagram below:



3 Form program structure based on the data structures from the previous step.

We have just one data structure. The program structure is therefore:



4 List and allocate operations

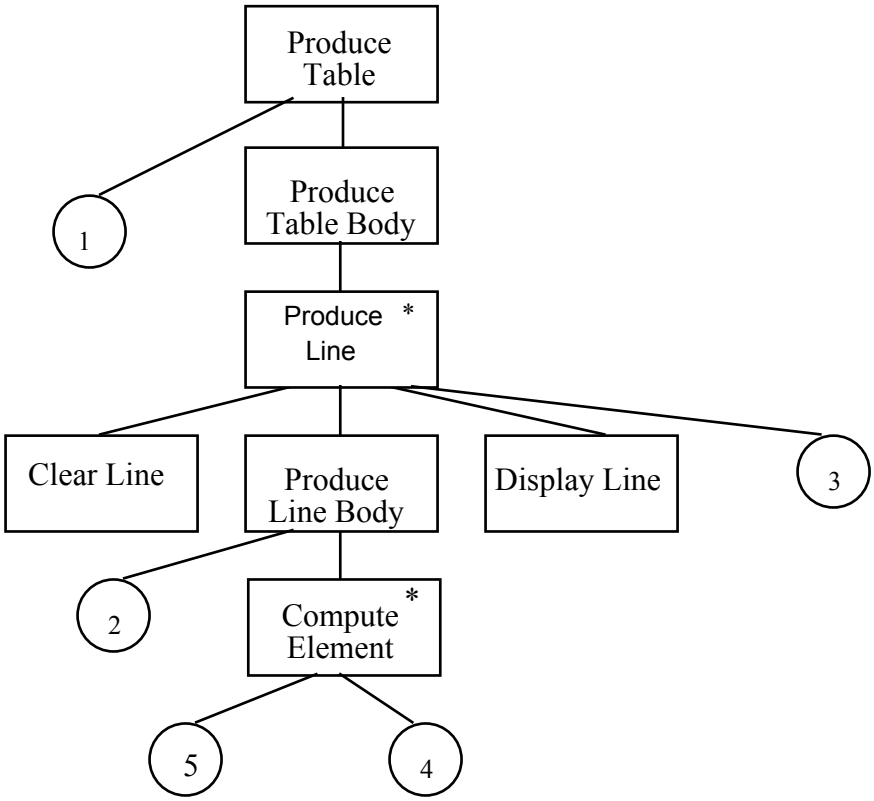
We note hat a line may contain either a number or a blank in any position. Representing a line by an array of integers, we will represent a blank by the integer value zero. Using bottom-up design, we define the procedures **clearline** and **displayline**.

We list the elementary operations needed to perform the task, and answer for each operation, "How often is it executed?" and "In what program component(s) does it belong?" The operations must be elementary statements of some programming language; we have chosen Pascal.

operation	how often?	where?
1 row-no := 1;	once	at start of program
2 col-no := 1;	once per line	in part that produces a line, at start
3 row-no := row-no + 1;	9 times	in part that produces a line
4 col-no := col-no + 1;	(row-no)-1 per line	in part that computes an element

5 line[col_no] := row_no*col_no once per element in part that computes an element

Having listed the operations, we next allocate them to our basic program structure to obtain an elaborated program structure. In order to accommodate the allocation of operations to components, we will almost always have to add new components, as we see in the structure diagram below, where we have added the components, "Produce Table Body" to allow for initializing row-no to 1, "Produce Line Body" to accommodate the operations before and after producing the elements in a line, and "produce-Element" to allow for incrementing the column prior to computing the next element.:



(5) Code program from structure diagram or structure text.

The structure text corresponding to the structure diagram above is given below:

```
Produce-Table   seq
    row-no := 1;
    Produce-Line   iter <while row-no <> 10>
        clearline;
        Produce-Line-Body
            col-no := 1;
            Produce-Element  iter <while col-no <> row-no>
                line[row_no] := col-no * row-no;
                col-no := col-no + 1;
            Produce-Element   end
        Produce-Line-Body
```

```
        displayline;     {print a line}
    Produce-Line  end
     row-no := row-no + 1;
Produce-Table   end
```

Note that we do not need to make explicit sequences within other components--thus, there is no structure text corresponding to the sequences "Produce Line ", "Produce-Line-Body" and Produce-Element"

The program text, which appears at the end of section 1.2, is easily coded from either the structure text or structure diagram.

3.2 Single Read-ahead Rule

Our multiplication table example involved no reading, only writing the lines of our table. The same output is produced each time. Most interesting programs are based on reading data from a serial file whose contents vary from one execution of the program to the next, so that different output is generated. We will see, moreover, that many awkward problems yield to treatment as problems in serial file processing, although at first glance they appear to be nothing of the kind.

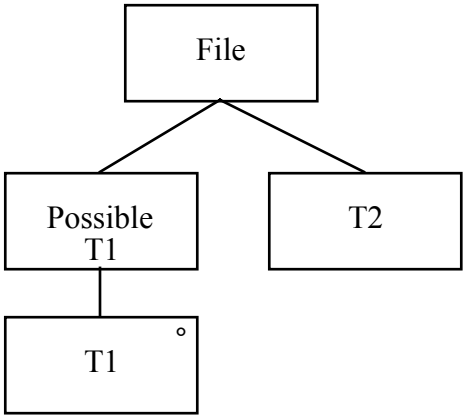
Suppose we have a file, F, consisting of two records, T1 and T2. It can be processed by a program with the structure below[6]:

[6] Adapted from Jackson [1], pp. 52-54.

```
P  seq
     P1  seq
         read;
         processT1;
     P1  end
     P2   seq
         read;
         processT2;
     P2   end
P  end
```

Here we have allocated the read operations at the start of each component that processes a record.

Suppose our file specification changes so that we may or may not have a T1 record at the start of the file, but will always have a T2 record. The data structure is thus:



Since only the specification for the part that processes a T1 record has been changed, one would suppose we could modify our program by changing only the P1 component. But this is easier said than done. Clearly, we cannot put the first read command in the component that processes a T1 record, because the condition test for the presence of a T1 record occurs at the start of the selection and depends on the T1 record already having been read. Thus, the first read must occur before component T1. The same would be true if we had an iteration, since the condition test comes at the start of the iteration. So, we will put the initial read prior to any component that uses the record. The record will then be available for any component that may need it. Putting the initial read at the beginning and leaving the second read operation at the start of the P2 component, we obtain the program structure below:

```
P  seq
     read;  {1st record is available to component P1}
   P1  seq
     POSST1  sel  <T1 present>
          processT1;
     POSST1  or    <T1 absent>
          ;  {null action}
     POSST1  end
   P1   end
   P2  seq
     read T2;
     processT2;
   P2  end
P  end
```

Our initial read is prior to the condition test for T1. If the T1 is absent, the T2 record is already present--the read in component P2 is not needed and will read beyond the T2 record. We only wish to read a second time if we have a T1 record. So, we are led to position the second read immediately after the processing of T1 as shown below:

```
P  seq
     read;          {initial read}
   P1  seq
     POSST1  sel  <T1 present>
          processT1;
          read;    {read-ahead}
     POSST1  or    <T1 absent>
          ;     {null action}
     POSST1  end
   P1   end
   P2  seq
     processT2;
   P2  end
P  end
```

We can generalize our strategy above with following rule:

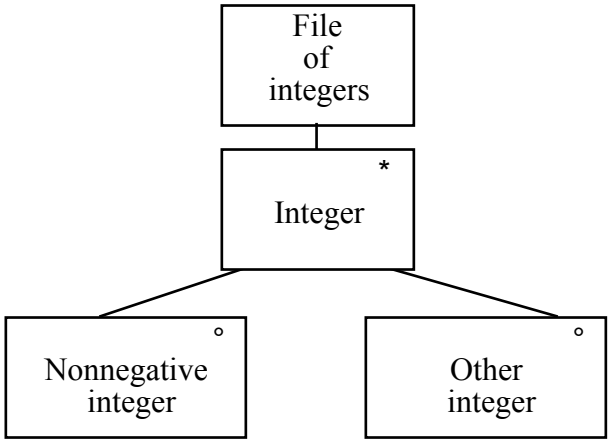
Single Read-ahead rule: Place the initial read immediately after opening a file, prior to any component that uses a record; place subsequent reads in the component that processes a record, immediately after the record has been processed.

The effect of the read-ahead rule is to have the next record (if any) available at the start of any component that may process it. We will see later that we sometimes need to have more than one record available at the start of a component; in this case we will need a multiple read-ahead rule.

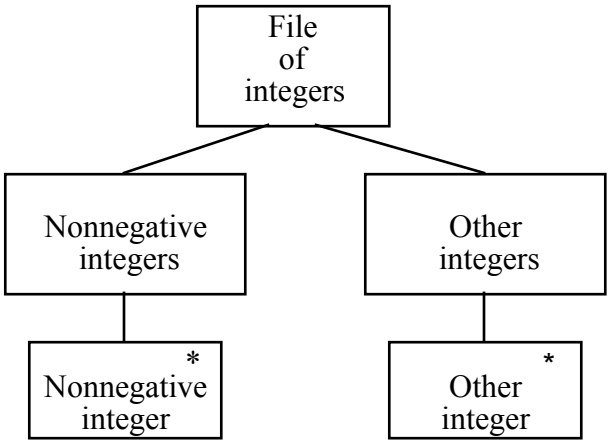
3.2.1 Pascal file processing: non-text files

Let us consider the following problem: A file of integers begins with a sequence of nonnegative integers whose sum we are asked to compute.

We might model our input file with the structure:



But, this structure doesn't tell us what the problem states, namely, that the file all of the nonnegative integers come before any positive integer. The structure we have shown only shows that any integer may be nonnegative or negative. Clearly, the correct structure of the input file is:



The structure text corresponding to this appears to be the following:

```
computeSum   seq
    reset(f);                  {f^ accesses the first integer, if any}
    sum := 0;
    Compute-nonnegative  iter <while not eof(f) and (f^>= 0)>
      sum := sum + f^;      {process the current integer}
```

```
        get(f);                 {read-ahead}
    Compute-nonnegative  end
    Other-integers iter <while not eof(f)>
        ;
    Other-integers end
    print "Sum= ", n;
    close(f);
computeSum   end
```

We can delete the block "Other-integers" since we can ignore the rest of the file once a negative integer has been read.

Unfortunately, the compound condition

not eof(f) **and** (f^>= 0)

presents a difficulty because, when eof(f) is true, f^ becomes undefined, and the relation

(f^>=0)

cannot be evaluated. In effect, we need to test first for the existence of a file component and, then, only if it exists, test its value. To express the semantics exactly, we need to express the condition with an expression like:

not eof(f) **and** (**if not** eof(f) **then** (f^>= 0)

but Pascal does not have the expressive power to do so. In order to avoid this difficulty, let's use a variable, n, instead of the buffer variable. We would normally initialize n explicitly at the beginning of the program, but we will assume that our Pascal compiler initializes all integer variables for us. Our structure text then becomes:

```
computeSum   seq
    reset(f);               {f^ accesses the first integer, if any}
    sum := 0;
    Compute-nonnegative  iter <while not eof(f) and (n >= 0)>
        sum := sum + f^;    {process the current integer}
        get(f);             {read-ahead}
    Compute-nonnegative  end
    print "Sum= ", n;
    close(f);
computeSum   end
```

We realize, however, that we have not assigned the first integer value to n . We must assign f^ to n, assuming f^ is defined. Similarly, we note that we must assign the new value of f^, if it exists, to n after the get(f) operation. Incorporating these two changes, we are thus led to the structure text below:

```
computeSum   seq
    reset(f);                  {f^ accesses the first integer, if any}
    sum := 0;
    assign-n  sel  not eof(f)
        n := f^;
    assign-n  end
    Compute-nonnegative  iter <while not eof(f) and (n >= 0)>
        sum := sum + f^;   {process the current integer}
        get(f);            {read-ahead}
        assign-n  sel  not eof(f)
            n := f^;
        assign-n  end
    Compute-nonnegative  end
    print "Sum= ", n;
    close(f);
computeSum   end
```

The Pascal program corresponding to the structure text given above is shown below:

```
program computeSum (input, output);
    const
        fname = 'data place 52:Development:JSP.pas:fileOfIntegers';
    var
        f: file of integer;
        sum, n: integer;

begin
    reset(f, fname);          {first file component, if any, is accessible via f^}
    sum := 0;
    if not eof(f) then
        n := f^;              {first file component, if any, assigned to n}
    while not eof(f) and (n >= 0) do
        begin
            sum := sum + f^;  {process current file component}
            get(f);           {advance file pointer  to next file component}
            if not eof(f) then
                n := f^;      {assign next file component, if any, to n}
        end;
    writeln(' sum = ', sum);
end.
```

Looking at the structure text, we note that the single read-ahead rule is followed: the reset(f) instruction opens the file; assignment of the buffer file variable, f^ to n

constitutes the initial read; the get(f) together with the assignment of f^ to n constitute the read-ahead, and immediately follows the processing of the current file component.

We would prefer to use a higher-level form of input, replacing get(f) and the assignment of f^ to a variable with a procedure that combines the operations of advancing to the next file component and assigning it to a variable into a single command.

We may be used to the schema for Pascal file processing in which the read command is placed within the iteration, as in the program below which computes the sum of a file of integers:

```
program computeSum (input, output);
  const
    fname = 'data place 52:Development:JSP.pas:fileOfIntegers';
  var
    f: file of integer;
    sum, n: integer;

  begin
  reset(f, fname);
  sum := 0;
  while not eof(f)  do
    begin
        read(f, n);
        sum := sum + n
    end;
  writeln(' sum = ', sum)
end.
```

In our problem to compute the sum of an iteration of nonnegative integers, we need an iteration with a compound condition:

```
begin
    reset(f, fname);
    sum := 0;
    while not eof(f) and (n >= 0)) do
        begin
            sum := sum + n;
        end
```

Where do we place the read statements? We cannot place the read statement within the iteration, because we need to know the value of n at its outset. We try the following:

```
begin
    reset(f, fname);
```

```
read(f, n);
sum := 0;
while not eof(f) and (n >= 0)) do
    begin
        sum := sum + n;
        read(f, n)
    end
```

But we cannot place the read prior to the iteration, since if the file were empty, we would attempt to read past the end-of-file marker. Even if the file were not empty, the code above is incorrect since:

(i) If the file contains just one integer, it will never be processed. The initial read will assign it to n, but advance to the next file component, and cause eof(f) to be true at the start of the iteration;

(ii) The last file component will not be processed for the same reason. The read-ahead assigns the last value of f^ to n, and then advances the file buffer variable, setting eof(f) true before the last value has been processed.

Recall that the standard Pascal procedure, read(f, n), is equivalent to

```
n:= f^;
get(f);
```

We note that in our structure text for the problem to compute the sum of nonnegative integers, the order of operations was reversed: first we advanced the file buffer variable with get(f); then we assigned f^ to n, if it existed. However, we cannot place a get(f) at the start of our program, since reset(f) is obligatory and achieves the same result. So, in a higher-level read operation using both operations, we must assign f^ and then advance the file buffer variable. If we record the status of end-of-file before the file buffer variable is advanced, then we can be assured of processing the current value assigned to n, and avoid the difficulties noted in the schema using the standard Pascal read procedure. We redefine the read procedure below:

```
type
    intfile = file of integer;
var  eofbit: boolean;
...
procedure xread(var f: intfile; var n: integer);
begin
    eofbit := eof(f);
    if not eofbit then
        begin
            n := f^;
            get(f)
        end
end
```

The global Boolean variable, eofbit, reflects the status of the file after the current file buffer component has been assigned to n, e.g. "read", but before the file buffer variable has been advanced. Eofbit is used for any subsequent end-of-file testing instead of eof(f). The structure text incorporating the redefined read follows the single read ahead rule:

```
computeSum   seq
    reset(f);
    xread(f, n);                {initial read}
    sum := 0;
    Compute-nonnegative  iter <while not eofbit and (n >= 0)>
      sum := sum + n;
      xread(f, n);              {read ahead}
    Compute-nonnegative  end
    print "Sum= ", n;
    close(f);
computeSum   end
```

Note that:

(i) We can have an initial read prior to the iteration since read will not advance beyond the end-of-file marker;

(ii) If there is but one component in the file, eofbit is false following the initial read, and the current value of the component will be processed within the iteration; the invocation of xread following processing will assign true to eofbit;

(iii) Immediately after the last record is read, eofbit is false but eof(f) is true. Thus, the last record will be processed.

The Pascal program corresponding to the structure text is shown below:

```
program computeSum (input, output);
    const
    fname = 'data place 52:Development:JSP.pas:fileOfIntegers';
    type
        intfile = file of integer;
    var
        f: intfile;
        eofbit: boolean;
        sum, n: integer;

    procedure xread(var f: intfile; var n: integer);
    begin
        eofbit := eof(f)
        if not eof(f) then
        begin
            n := f^;
            get(f)
```

```
        end;
    end

begin
  reset(f, fname);
  xread(f, n);
  sum := 0;
  while not eofbit and (n >= 0) do
    begin
          sum := sum + n;
          xread(f, n);
    end;
  writeln(' sum = ', sum)
end.
```

Replacing the standard Pascal read procedure allows us to apply the single read-ahead rule, derived from the basic logic of condition testing prior to selection or iteration components, as a general solution for file processing. The details of the xread procedure, the result of bottom-up design, are properly hidden from the program--they are outside of the boundary of the model on which the program is based.

There are many cases where the standard Pascal schema--placing the reset procedure at the start of a program but placing the standard read procedure after the eof test as the first statement of the iterated component--works fine. But it fails in cases where there is a condition at the head of the iteration that depends on the current record's contents. Throughout this text, we will therefore enforce the single read-ahead rule.

Note: In COBOL, we have no such difficulties. The single read-ahead rule can be applied perfectly, as shown in the segment below for the problem to compute the sum of an iteration of nonnegative integers:

```
WORKING-STORAGE SECTION.
...
    02  IN-EOF PIC X VALUE SPACE.
        88  IN-EOF VALUE 'E'.
...
PROCEDURE DIVISION.
PROCFILE.
    OPEN INFILE.
    READ INFILE AT END MOVE 'E' TO IN-EOF.
    MOVE ZERO TO SUM.
    PERFORM COMPUTE-SUM UNTIL ((IN-EOF) OR (N .GE. 0)).
    DISPLAY "SUM =", SUM.
    CLOSE INFILE.
    STOP RUN.
COMPUTE-SUM.
```

```
ADD N TO SUM GIVING SUM.
READ INFILE AT END MOVE 'E' TO IN-EOF.
```

3.2.2 Pascal file processing: textfiles

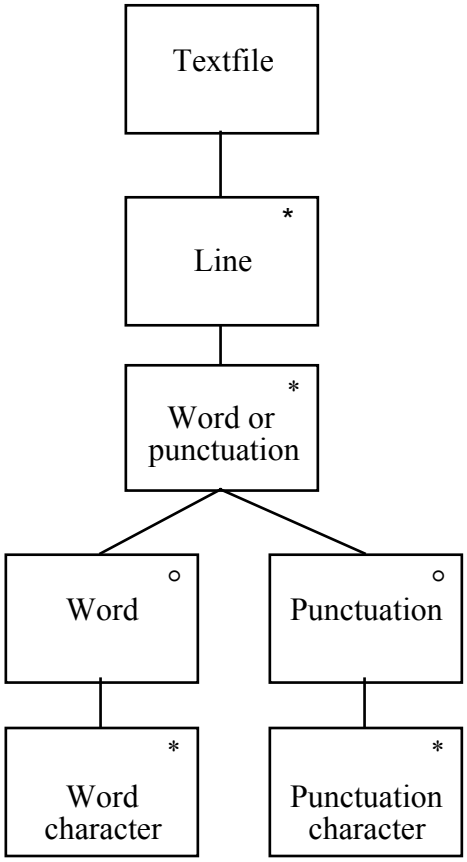
As another example, let us design a program to read a textfile consisting of some text and output each word of the input file as a separate line of output. A word is defined as any sequence of letters and apostrophes.

We may be tempted to begin programming by using an existing program, known to work correctly. This technique, used by many experienced programmers, is analogous to the method of fixed point iteration: like the mathematical method, our first approximation to the solution will be a guess, which we will subsequently successively refine. We take as our first approximation the standard structure text below for copying an input text file to an output file:

```
reset(f);  rewrite(g);
begin
    while not eof(f) do
    begin
        while not eoln(f) do
        begin
            read(f, ch);
            write(g, ch);
        end
        writeln(g); readln(f);
    end
    close(f);
    close(g)
end
```

In the structure text, the read procedure refers to the standard Pascal read procedure for textfiles.

The input file consists of lines of text. Each line consists of words alternating with punctuation. After some experimentation, we arrive at the following input file structure:



Within each line we have an iteration of word-characters or an iteration of punctuation-characters. As our first modification, we try the following:

```
reset(f);  rewrite(g);
begin
    while not eof(f) do
    begin
        read(f, ch);
        if (ch in word-char) then
            while not eoln(f) and (ch in word-char) do
            begin                          (read & write a word}
                write(g, ch);
                read(f, ch)
            end
            ...
            ...
        else                               (skip past punctuation}
        ...
        ...
        end
    close(f);
    close(g)
```

end

We must place the initial read before the condition test for a word-character, and we reverse the read and write within the iteration. This gives the familiar single read-ahead pattern. But, of course, with the standard Pascal read procedure, a quick analysis shows that this schema will not work:

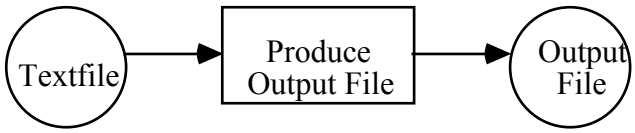
(1) If the input file consisted of the single word, "a", eoln will be true following the initial read, and so we will not process our single character file;

(2) The last character in the file won't be processed for similar reasons.

Evidently, as we saw in the previous section, we want to test for end-of-line after f^ is assigned to ch, but before advancing to the next file component. We could redefine our read procedure accordingly, and proceed further in our analysis. This is left as an exercise.

Instead, we will start afresh using JSP. The design steps are shown below:

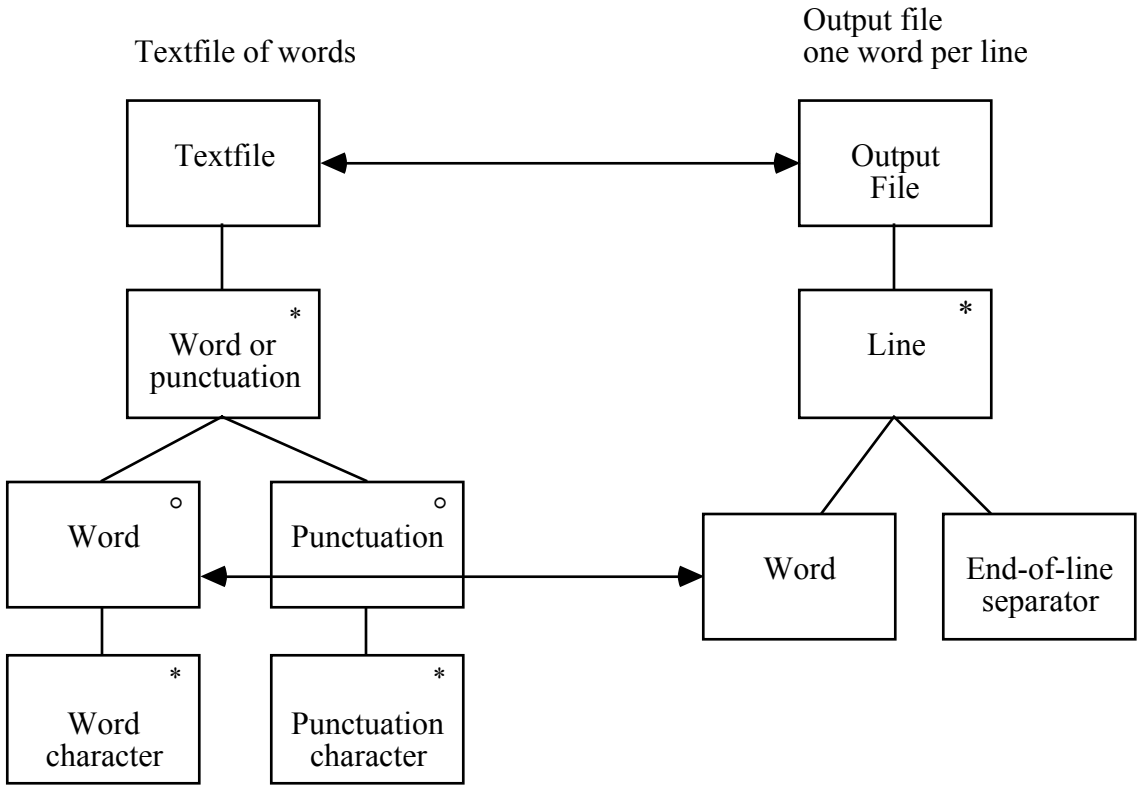
1 Draw a system diagram



2 Draw a data structure for each input and output file

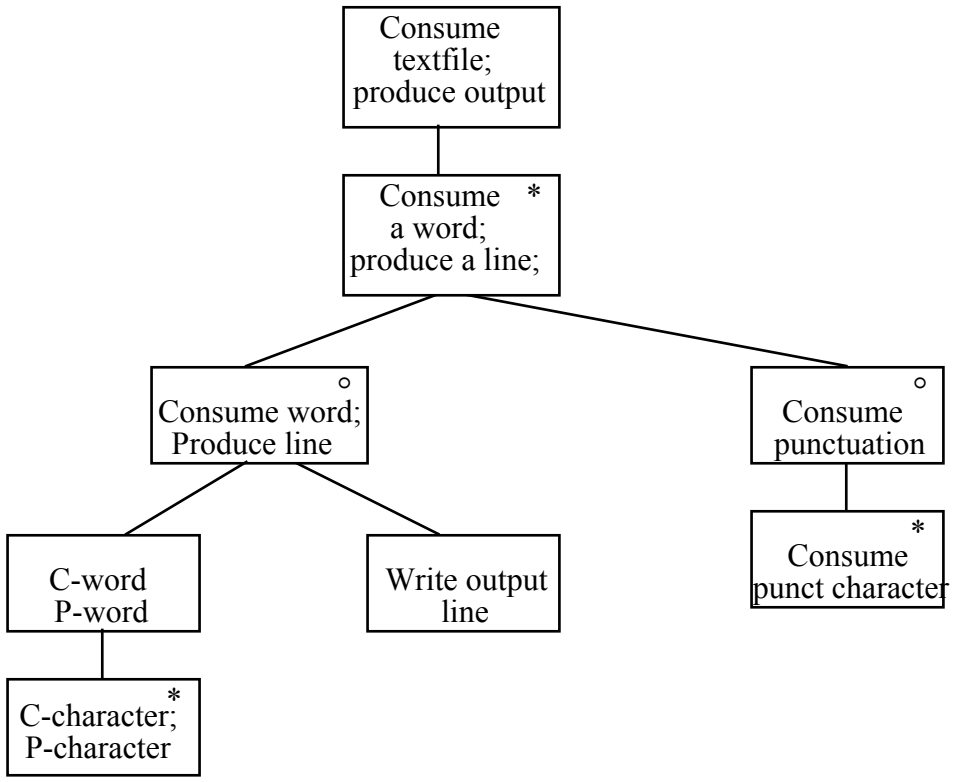
After discussing the problem with its originator, we satisfy ourselves that we need not concern ourselves with the line structure of the input--there won't be any need to keep track of the number of words per line in the input file, for example. Thus, we will model the problem without reference to the line structure of the input textfile. Instead, we will do some bottom-up design and design a procedure to read the next character from our input file. Since the specification requires one word per line on the output file, we must model the output file explicitly as an iteration of lines.

The structure of the input and output files are shown below together with correspondences between input and output components:



There is one output file for each input textfile; each word on the output file corresponds to a word in the input file; preceding and following each word, there may be one or more punctuation characters.

3 Draw a single data structure based on correspondences between the input and output data structures; this data structure forms the basic program structure which is shown below:



4 List the operations needed by the program, For each, ask "Where does it belong (in what program part?)" "How many times does it occur?" Allocate the operations to the basic program structure.

operation	how often	where?
1 reset(infile);	once	at start
2 rewrite(outfile);	once	at start
3 xread(infile,ch);	n times, where n = # words in textfile in component that consumes punctuation characters	immediately after reset; in component that consumes a word;
4 write(outfile, ch);	once per word	in part that outputs a character
5 writeln;	once per word	in part that outputs a line, after writing a word
6 close(textfile);	once	at end of program
7 close(outfile);	once	at end of program

The structure text corresponding to the program structure is shown below:

```
C-textfile   seq
    reset(infile);
    xread(infile, ch);
    rewrite(g);
```

```
C-textfile-body   iter  <while not eof>
    C-word-or-punct   sel  <word-char>
            C-word-char iter <while not eof and in word-char>
                write(g, ch);
                xread(infile, ch);
            C-word-char  end
            writeln(g);
    C-word-or-punct   alt <while not eof and not in word-char>
            C-punct-char  iter <while not eofbit and not in word-char>
            xread(infile, ch);
            C-punct  end
    C-word-or-punct   end
C-textfile-body   end
close(infile);
close(g);
C-textfile   end
```

Our structure text is intelligible: The component, C-word, consumes and outputs a word. The operation, writeln(g), skips to the next line, immediately after a word has been consumed and output. There is a component that skips past the punctuation between words.

In our structure text, the Pascal read procedure has been redefined using bottom-up-design to test for end-of-file and end-of-line after ch has been assigned the value of f^, if it exists, but before advancing to the next file component. The redefined procedure is shown below:

```
procedure xread(var f: text; var  ch: integer);   {read a character from a text file}
    begin
        eofbit := eof(f);
        if not eofbit then
            begin
                eolnbit := eoln(f);
                if not eolnbit then
                    begin
                        ch := f^;
                        get(f)
                    end
                else
                    begin
                        ch := ' ';     {if end-of-line, set ch to a blank indicating end
                                    {of word}
                        readln(f);    {and skip to beginning of next line}
                    end;
            end;
    end;
```

In our xread procedure, in fact, we don't need to model the line structure of the input text file. We could as well use the following procedure, akin to our procedure to read from a non text file:

```
procedure xread(var f: text; var  ch: integer);   {read a character from a text file}
begin
    eofbit := eof(f);
    if not eofbit then
        begin
            ch := f^;
            get(f)
        end
end;
```

5 Translate the program structure into text, specifying the conditions for iteration and selection. The Pascal program is shown below; the text of the xread procedure, is omitted, and the program writes to the standard output file:

```
program TextfileofWords (input, output);
    const
        infilename = 'data place 52:Development:JSP.pas:TextfileofWords.txt';
    var
        f: text;
        ch: char;
        eofbit, eolnbit: boolean;

begin
    reset(f, infilename);
    xread(f,ch);                                        {initial read-ahead}
    while not eofbit do
        begin
            if (ch in ['a'..'z', 'A'..'Z', '']) then   {word}
                begin
                    while not eofbit and (ch in ['a'..'z', 'A'..'Z', '']) do
                        begin                          {consume and output a word}
                            write(ch);
                            xread(f, ch)
                        end;
                    writeln                {skip to beginning of next line}
                end
            else
                while not eofbit and not (ch in ['a'..'z', 'A'..'Z', '']) do
                    xread(f, ch);                      {skip punctuation}
```

```
        end;
    close(f);
end.   {program}
```

Note: In the programming language C, the handling of text files is much simpler than in Pascal, and the single read-ahead rule can be applied without difficulty, as shown in the program below to copy an input file to an output file character-by-character:

```
#include <stdio.h>
/* copy input to output */
{
    int c;

    c = getchar();        /*  initial read */
    while (c != EOF)  {
        putchar(c);
        c = getchar();  /* read ahead */
    }
}
```

Ex. (iii) A file consists of a sequence of positive integers followed by a sequence of negative integers. Compute the sum of both sequences. Write the program assuming (a) input file of integers, and (b) input text file. How does the read procedure you define differ in the two cases?

(iv) A file contains sets of data. Each set consists of a header record, followed by a number of data records. Each header record contains an integer that tells how many data records will follow. Each data record contains two real numbers. Write each set of data as a separate output file with the name, FILEn, where n is a consecutively numbered integer (1,2,3,...).

(v) A file consists of sets of data. Each set consists of a header record followed by a number of data records, as in problem (i) above. Each data record contains three real numbers, and the three sets of real numbers form three distinct data sets. Write each data set to a separate output file, putting the 1st number from an input data record to the 1st file, the 2nd number from an input data record to the 2nd file, and the 3rd number from an input data record to the 3rd file. (Thus, each output file contains data records consisting of just one real number.) Name each output file with the name, FILEmn, where m = a consecutively numbered data set (1,2,3,...) and n is 0,1 or 2 depending on whether a data record contains the 1st, 2nd or 3rd input real number on the input file.

(vi) Using the schema below to input a textfile using the standard read(f, ch) procedure, try to rewrite the program to produce lines of output containing one word per

line from an input file of text. Is the program intelligible? Is there a component that reads and writes a word?

```
reset(f);  rewrite(g);
begin
    while not eof(f) do
    begin
        read(f, ch);
        if (ch in word-char) then
            while not eoln(f) and (ch in word-char) do
            begin                       (read & write a word}
                write(g, ch);
                read(f, ch)
            end
        ...
        else                            (skip past punctuation}
        ...
        ...
    end
    close(f);
    close(g)
end
```

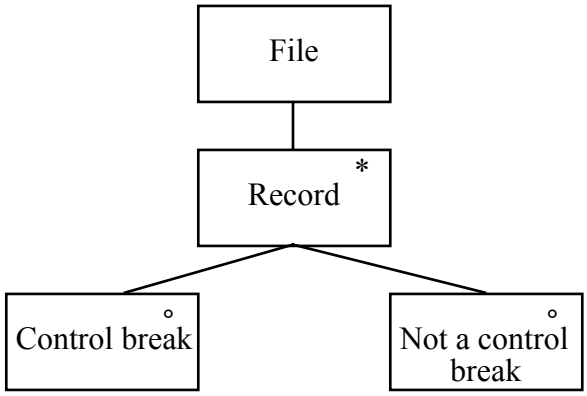
(vii) In addition to outputting words from an input text file, one per output line, your program should report a count of the number of words per line and the number of characters per word. Using JSP, rewrite the program, modeling the line structure of both input and output text files.

4. Basic Design Method: Multiple Data Structures

Let us continue to discuss the basic design method introduced in the last chapter, concentrating on situations where multiple inputs and/or output files are used, and where correspondences in these structures must be examined in constructing the program structure.

4.1 Processing record sets in a sequential file

A problem that occurs frequently in data processing involves accumulating totals in a serial file for a set of records having the same record identifier. Frequently, programs are based on the incorrect structure shown below:



The resulting program is more complex, harder to understand, and certainly harder to maintain than a program based on the correct structure, as the next two sections recount.

4.1.1 Getting It Wrong--A Cautionary Tale[7]

Consider the following problem: A warehouse records a transaction for every item received into or issued out of the warehouse. At the end of the day, the transaction file is sorted by item number, and a "Daily Net Movement Summary" showing the net movement of each item into or out of the warehouse is produced. The format is shown below:

Daily Net Movement Summary

A12345	40
A23456	-30
.....	
Z13579	25

[7] This section is adapted from an article with the same title by Michael Jackson found in Cameron [1]

End Summary

Experienced programmers have seen this type of problem, and even those who don't know about it can find its solution in that very fine book, Principles of Program Design by Michael A. Jackson!

This story is about a novice programmer who hadn't seen the problem before. He sketched out a top-down solution to the program and arrived at the following pseudocode:

```
begin
    reset transfile;  read transfile; writeln(' Daily Net Movement Summary');
    while not end-of-file do
        if new group then
            end old group
            start new group
        else
            process record
        endif
        read transfile
    endwhile
    writeln('End Summary');
    close file
end
```

Of course, some of you more experienced programmers will see immediately that something isn't quite right. Our novice, however, proceeded to code the program in Pascal and run it.

The program produced the following output:

```
Daily Net Movement Summary
                    0
A12345             40
A23456            -30
.....
Z13579             25
End Summary
```

What was that first line containing 0, the novice wondered? After a little thought, he realized it was due to ending an old group before the first group. He had heard something about a first-time-switch from a colleague down the corridor, so he added the following fix to his original structured code:

```
begin
    reset transfile;  read transfile; writeln(' Daily Net Movement Summary');
```

```
    sw <-- false;
    while not end-of-file do
        if new group then
            if (sw) then
                end old group
            else sw <--true
            start new group
        else
            process record
        endif
        read transfile
    endwhile
    writeln('End Summary');
    close file
end
```

Running the program again, he was gratified to see the first line disappear from the output, and soon after, the program was put into production. After a few weeks, a clerk from the user department came to our programmer and said, "Look, there's no total for the last group!" Of course, you experienced programmers would have known this, since initially the end-group and start-group operations had been paired, but the introduction of the first time switch removed one of the end-groups. As a result, there was a group started but not ended, and indeed, this was the last group in the file. So our novice programmer made the obvious correction, arriving at the code shown below:

```
begin
    reset transfile;  read transfile; writeln(' Daily Net Movement Summary');
    sw <-- false;
    while not end-of-file do
        if new group then
            if (sw) then
                end old group
            else sw <--true
            start new group
        else
            process record
        endif
        read transfile
    endwhile
    end last group
    writeln('End Summary')
    close file
end
```

Pretty soon, on Thanksgiving day, in fact, the program was run, and the output produced was quite unexpected:

```
                Daily Net Movement Summary

                        0

                End Summary
```

Obviously, something was wrong--the unwelcome 0 line had reappeared--why? Then, it became clear. There had been no transactions, and so there were no groups. The mystery line was the result of ending the last group--only there wasn't any last group.

Our novice programmer was tempted to do something with the first-time switch, sw, but, having heard something about "defensive programming" decided against this strategy. Defensive programming appears to be the theory, that, when making changes to a program, whose possible side effects aren't understood, one ought not to tamper with parts of the program that appear to work. So, our novice decided to add a second switch, sw2, as shown below:

```
begin
    reset transfile;  read transfile; writeln(' Daily Net Movement Summary');
    sw <-- false;  sw2 <-- false;
    while not end-of-file do
        if new group then
            if (sw) then
                end old group
            else sw <--true
            start new group
        else
            process record
            sw2<--true
        endif
        read transfile
    endwhile
    if (sw2) then
        end last group
    writeln('End Summary')
    close file
end
```

After recompiling his program, our programmer ran it against an empty transaction file with the desired result:

```
                Daily Net Movement Summary
                End Summary
```

Clearly, our programmer's trials were at an end. The program ran again for the next 18 months.

Then one day, a clerk from the user department came in and said, "Look, the last group has been left off the printout again!" The diagnosis was easy. An old, incorrect version of the program had been run. But, a couple of days of intensive detective work showed that this was not the case--the correct version of the program had been run! The only possibility was a transient hardware error or some random problem with the operating system.

Indeed, this seemed to have been one of those rare occasions. After all, the problem hadn't occurred before--at least, not since the 'end last group' fix had been added to the program.

Then, a funny thing happened. The production clerk came in one day, and said to our programmer: "Do you remember that last group that was left off the printout? Well, I got hold of the transaction input file, and found there were 843 transactions." "So what?" replied the programmer. "Well, there were 842 group totals on the printout." "I don't see that that's relevant", said the programmer, "but thank you for mentioning it to me."

That night he took the program home, and studied it carefully. Suddenly it dawned on him! Of course! If there were 843 transactions and 843 totals, then each group contained exactly one record: so the condition 'new group' would be true on every card, and it would always be the 'if'; that was executed and never the 'else'. But the instruction to set sw2 to true was only in the 'else' clause! So sw2 was never set to true, and the last group was never ended.

Our programmer, having diagnosed the problem, solved it easily by adding a statement to set sw2 true to the first clause of the 'if-else' as shown below:

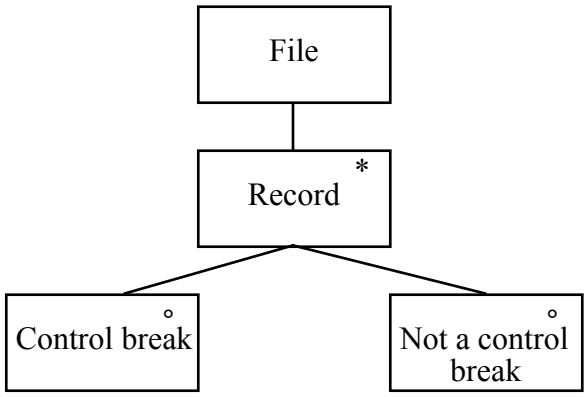
```
begin
    reset transfile;  read transfile; writeln(' Daily Net Movement Summary');
    sw <-- false;  sw2 <-- false;
    while not end-of-file do
        if new group then
            if (sw) then
                end old group
            else sw <--true
            start new group
            sw2<--true
        else
            process record
            sw2<--true
        endif
        read transfile
```

```
    endwhile
    if (sw2) then
        end last group
    writeln('End Summary')
    close file
end
```

Obviously, the program is now perfect and correct. But, of course, it has only been running for three months in its new form, and I will keep you posted if there are any new surprises.

* * * * *

And the moral? It's this: the structure of the program produced by our novice programmer is one we often see:



and this structure is wrong. Not inferior; not inelegant; just plain wrong. The difficulties were all caused by the 'end group' instructions. Now, how often should we end a group? Why, once per group! Where, in this program, is there a component that processes each group? There is no such component, and therefore the 'end group' instructions cannot be correctly allocated to the program structure. That's what all the difficulty was about.

Of course, all of us experienced programmers never make this kind of mistake. But, I know many very experienced programmers who make just this sort of mistake on bigger and more obscure programs; this type of mistake--having the wrong program structure--occurs frequently!

A second moral is this: Avoid first time switches like the plague! Each adds two possibilities to consider, and a program with n switches adds 2^n possibilities, and becomes quite unintelligible. First time switches are usually introduced because a program's structure is not correctly understood--they thus conceal rather than make explicit a program's true structure.

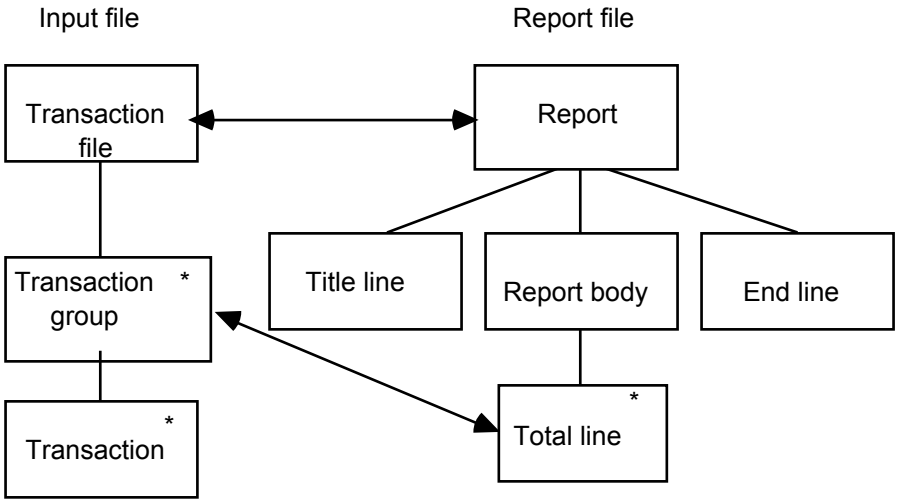
4.1.2 Getting It Right

Here is how to design the program correctly using JSP:

1 Draw a system diagram

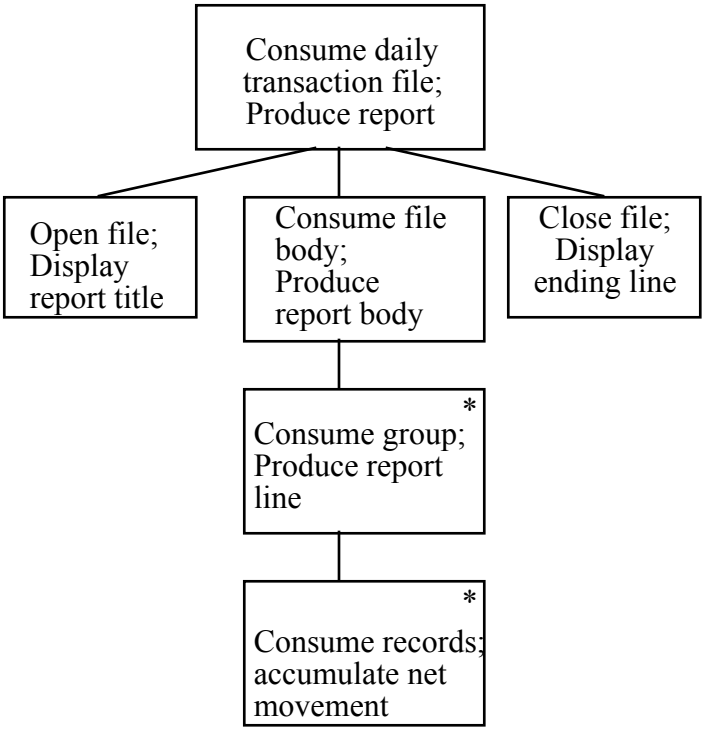
When the system diagram is self-evident, we can omit it. We do so here.

2 The structure diagram below shows the input and output structures together with their correspondences:



There is one "Daily Net Movement Summary Report" per daily transaction file; for each group of transactions recording receipts and issues for a single item, one report line is produced.

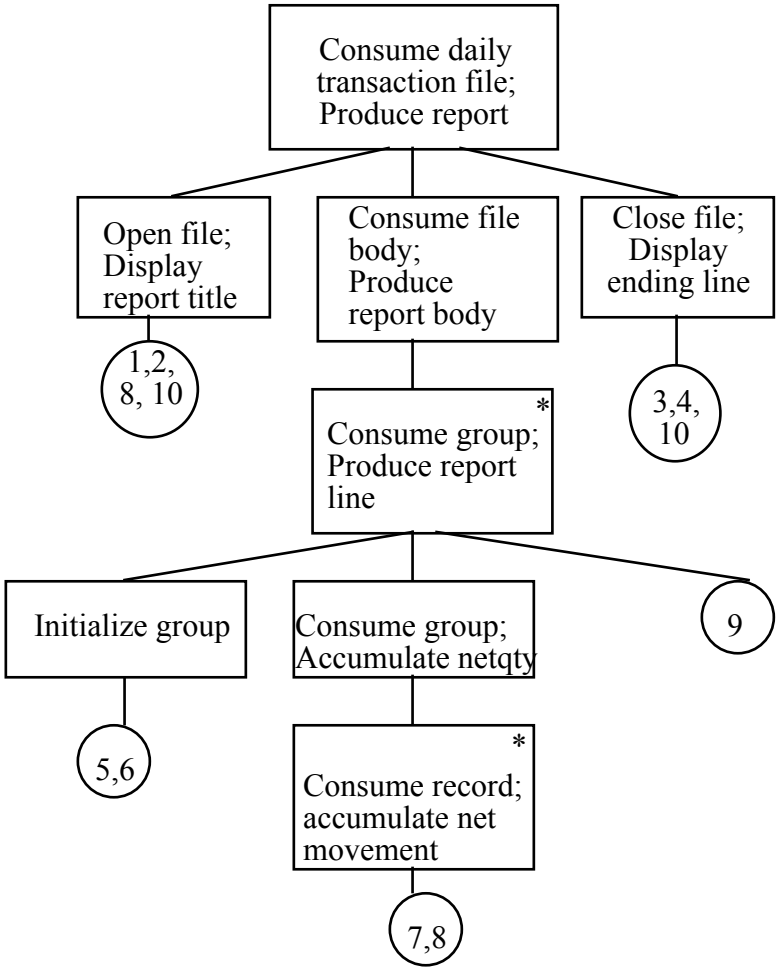
3 The single program structure based on correspondences between the input and output data structures is shown below:



4 The operations needed by the program are listed and allocated to the basic program structure below:

operation	where?	how often?
1 reset(f)	once	at start
2 writeln(' Net Daily Movement Summary')	once	at start
3 writeln(' End Summary')	once	at end
4 close(f)	once	at end
5 groupid := rec.itemno	once/group	in component to process a group
6 netqty := 0	once/group	in component to process a group
7 netqty := netqty + rec.qty	once/transaction	in component to process a record
8 xread(f,rec)	once per record	at start; in component that processes each record
9 write(' ', groupid, netqty)	once per group line	in component to produce a report
10 writeln	once after title; once before ending line	at start; at end

The allocation of these operations to the basic program structure to form an elaborated program structure is shown below:



5 Finally, the translation of the program structure into structure text, with the conditions for iteration and selection specified, is shown below:

```
P-netmovement   seq
    reset(infile);
    xread(infile, rec);
    writeln(' Daily Net Movement Summary');
    writeln;
    C-file-body   iter  <while not eofbit>
        C-group  seq
            groupid := rec.itemno;
            netqty := 0;
            C-groupbody   iter <while not eofbit and (groupid = rec.itemno)>
            netqty := netqty + rec.qty;
                xread(infile, rec);
            C-groupbody  end
            writeln(' ', groupid, netqty);
        C-group    end
```

```
        C-file-body   end
        close(infile);
        writeln;
        writeln(' End Summary')
    P-netmovement  end
```

The translation of the structure text into Pascal with a suitably redefined read procedure is left as an exercise.

4.2 Group-id Rule

The accumulation of totals over a record set based on a record identifier on which the records are sorted is a typical example of hierarchical record processing. Each level of the hierarchy contains an iteration of records. The following design rule applies:

> **Group-id Rule**: If a structure contains a component that is a group of records all having the same value of a (usually sorted) identifier, then there must be an operation that stores the value of the identifier. This operation should be allocated once per group, at the beginning of the group.

In the structure text to produce the "Daily Net Movement Summary", the group-id variable stores the value of each group, and is allocated at the start of the component to process a group in accordance with the group-id rule.

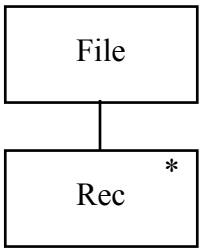
4.3 Collating

A class of data processing problems concerns the correspondence among input files rather than between input and output files. Collating, or "matching" sequential files is such a problem.

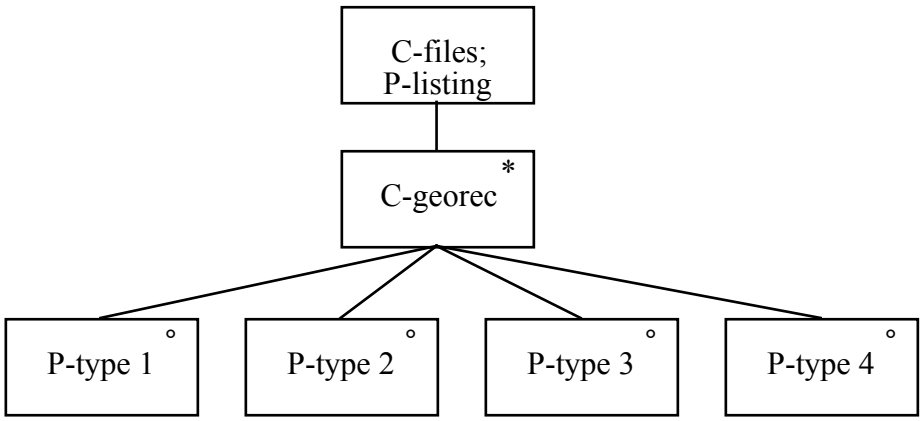
Consider the following problem from census processing involving three files, each sorted by the same geographic identifier or geocode, a composite key consisting of concatenated country, province, district, and village identifiers. The first file is a reference file containing the set of village identifiers together with their names; the second file contains population data for each village, while the third data file contains housing data for each village. The problem is to produce a listing of geocodes classified into the following four types:

1 - geocodes from the reference file with no corresponding data on either population or housing file;
2 - geocodes with matching population and housing data;
3 - geocodes with matching population data (but no housing data);
4 - geocodes with matching housing data (but no population) data

Each of our files has the same structure, namely:



But what are the correspondences on which our program structure is based? Clearly, each type in our listing requires that we examine the geographic code in all three files. The program component to process any of the types in the listing doesn't correspond to any single file structure. The program structure may be depicted as shown below:



Translation of this program structure gives the following structure text:

```
P-geomatch   seq
    reset(georef); reset(pop); reset(hous);
    readgeo(georef, georec); readpop(pop(pop, poprec); readhous(hous, housrec);
    writeln(' Geographic code analysis  ');
    writeln;
    C-georef   iter  <while not eofref)>
        C-type  sel  < georec.geocode = poprec.geocode=housrec.geocode >
            type := 1;
            writeln(' ', type, ' '; georec.geocode, georec.geoname, poprec.pop,
            housrec.hhlds);
            readpop(pop, poprec); readhous(hous, housrec);
        C-type  alt  <  poprec.geocode = georec.geocode < housrec.geocode >
            type := 2;
            writeln(' ', type, ' ', georec.geocode, georec.geoname, poprec.pop);
            readpop(pop, poprec);
        C-type  alt  <  poprec.geocode > georec.geocode = housrec.geocode >
        type := 3;
            writeln(' ', type, ' ', georec.geocode, georec.geoname, housrec.hhlds);
```

```
            readhous(hous, housrec);
        C-type  alt  <  poprec.geocode > georec.geocode < housrec.geocode >
        type := 4;
            writeln(' ', type, ' ', georec.geocode, georec.geoname);
        C-type  end
        readgeo(georef);
    C-georef    end
    close(georef); close(pop); close(hous);
    writeln;
    writeln(' End Listing')
P-geomatch  end
```

Note that the single read-ahead rule has been used; in particular, note the test of the global variable, eofrec, instead of eof(georef) in the iteration over the village reference file. The Pascal program is shown below:

```
program geomatch (input, output);
const
    geoname = 'data place 52:Development:JSP.pas:geo';
    popname = 'data place 52:Development:JSP.pas:pop';
    housname = 'data place 52:Development:JSP.pas:hous';
type
    geocode = packed array[1..10] of char;
    refrec = record
        refcode: geocode;
        refname: packed array[1..35] of char;
        end;
    poprec = record
    refcode: geocode;
    persons: integer;
        end;
    housrec = record
    refcode: geocode;
    hhlds: integer;
        end;
    reff = file of refrec;
    popf = file of poprec;
    housf = file of housrec;

 var
    matchtype: integer;
    eofref, eofpop, eofhous: boolean;
    georef: reff;
    pop: popf;
    hous: housf;
    recref: refrec;
```

```
    recpop: poprec;
    rechous: housrec;

procedure readgeo (var georef: reff; var recref: refrec);
begin
 eofref := eof(georef);
 if not eof(georef) then
  read(georef, recref);
end;

procedure readpop (var popref: popf; var recpop: poprec);
begin
    eofpop := eof(pop);
    if not eof(pop) then
        read(pop, recpop);
end;

procedure readhous (var housref: housf; var rechous: housrec);
begin
    eofhous := eof(hous);
    if not eof(hous) then
        read(hous, rechous);
end;

begin
    reset(georef, geoname);
    reset(pop, popname);
    reset(hous, housname);
    readgeo(georef, recref);
    readpop(pop, recpop);
     readhous(hous, rechous);
    writeln(' Geographic code analysis  ');
    writeln;
    while not eofref do
        begin
        if (recref.refcode = recpop.refcode) and
             (recpop.refcode = rechous.refcode) then
            begin
            matchtype := 1;
            writeln(' ', matchtype, ' ', recref.refcode, recref.refname,
            recpop.persons, rechous.hhlds);
            readpop(pop, recpop);
            readhous(hous, rechous);
            end
            else if (recpop.refcode = recref.refcode)
                and (recref.refcode < rechous.refcode) then
```

```
                    begin
                    matchtype := 2;
                    writeln(' ', matchtype, ' ', recref.refcode, recref.refname,
                    recpop.persons);
                    readpop(pop, recpop);
                    end
                    else if (recpop.refcode > recref.refcode)
                        and (recref.refcode = rechous.refcode) then
                        begin
                        matchtype := 3;
                        writeln(' ', matchtype, ' ', recref.refcode, recref.refname,
                        rechous.hhlds);
                        readhous(hous, rechous);
                        end
                        else if (recpop.refcode > recref.refcode)
                            and (recref.refcode < rechous.refcode) then
                            begin
                            matchtype := 4;
                            writeln(' ', matchtype, ' ', recref.refcode,
                            recref.refname);
                            end;
            readgeo(georef, recref);
        end;
        close(georef);
        close(pop);
        close(hous);
        writeln;
        writeln(' End Listing')
    end.
```

Exercises

(i) Translate the structure text for the net movement summary report into Pascal, redefining a suitable read procedure to process the input transaction file.

In exercises (ii)-(iv), one person or team uses JSP to design a program corresponding to the initial specifications given; then, to produce the modifications given to the initial problem, a second person or team modifies the initial design, producing a modified report.

(ii) A student grade file contains the grades for students taking courses at the university. Each record contains student-id, student name, course-id, course name and grade. The file is sorted by course-id within student-id. A report, "Student Grades", is desired that contains for each student his or her grades and the average for all the courses taken during the semester. In addition, the report prints a summary containing the number of students in all courses, and average grade for all courses.

Modification: Each record contains a code that indicates the subject that a student is specializing in. The file is sorted by course-id within student-id within faculty-id. Produce a summary line for each faculty id containing the number of students specializing in the faculty and their average grade.

(iii) A population census file is sorted by person-id within household-id. Each record contains household-id, person-id and gender coded 1 for female and 2 for male. Print report lines containing the household-id and the number of males and females in each household. Print a summary line with the number of males, females and households in the file, together with the average household size.

Modification: Each record also includes enumeration-area (EA) identification code, and the file is sorted by person-id within household-id within enumeration area. Modify the program to include in the report the number of males and females in each EA, together with average household size in each EA.

(iv) A program has to process a single input file and produce one report. The input file is sorted by customer-id. Each record contains a product identifier and quantity field, which indicates how much of the product that a customer has ordered. There are two types of customer, discount customers and normal customers, distinguished by a code of "D" or "N" in a record. A database can be directly accessed to give the cost per unit of a product
based on the product identifier code and the outstanding debt of a customer, based on the customer-id code. "D" type records also have a discount field, that represents the percentage discount the customer gets on all items.

The summary report consists of one line for each normal customer, two lines for each discount customer, and two overall summary lines. The normal customer summary line has the total value of the transactions, the old and the new debt. The first
line of a discount summary has the total value of the transactions, the discount level, and the discounted value. The second line has the old debt and the new debt. The overall summary totals the debts and values.

Modification: Each header record contains an area code in addition to customer-id, and the input is sorted on customer-id within area code. Modify the previous report to include a
summary line for each area detailing the total value and total discounted value of the transactions in that area.

Exercises (v)-(vii) could be solved profitably using paired-problem-solving.

(v) [Part 1] In the collating problem discussed in Section 4.2, show the structure text and Pascal code using the standard Pascal read procedure. Discuss the resulting program in terms of its simplicity and intelligibility.
[Part 2] Now suppose that the population and housing files each could contain a group of data records for any village code. Design a program using JSP to produce the required listing of four types.

(vi) [Part 1] In the collating problem discussed in Section 4.2, suppose that the village reference file may contain geographic coding errors: in other words, there may be unmatched geocodes on both the population and housing files. Write the structure diagram and structure text for a program to produce a listing of geocodes classified into the six resulting types of matching.
[Part 2] Produce the same listing as in Part 1 assuming that the population and housing files may contain a group of records for any geographic code.

(vii) [Part 1] In addition to the village reference, population and housing files in the collating problem discussed in Section 4.2, there is a mapping file that contains the geographic features of each village. Write the structure diagram and structure text for producing a listing of geocodes for every geocode, assuming (a) no errors on the village reference file and (b) geographic coding errors on the village reference file. How many types are there in each case?
[Part 2] Produce the same listing as in Part 1, assuming that the population and housing files may contain a group of records for any geographic code.

5. Errors and Invalidity

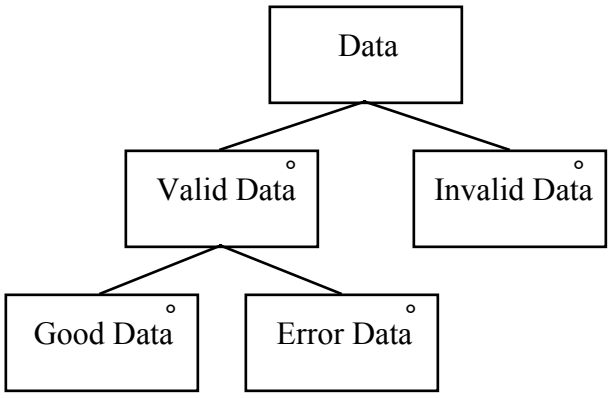
5.1 Introduction

Error processing accounts for a high proportion of program code in information systems. In on-line transaction processing, for example, the program that processes on-line transactions includes a substantial dialogue with the user in which user errors are detected, diagnosed, and the user prompted to reenter data. And in updating a set of entities (for example, warehouse stock items) with current activity recorded on a transaction (a stock shipment), the program that maintains the data must detect errors such as a transaction for which no corresponding entity exists (an invalid part number), or a transaction which is invalid for the corresponding entity (an order for a quantity of a part that exceeds its current availability).

We may specify validity at different levels: for a field (for example, numeric, within a specified range, etc.); among different fields within a single transaction or record (consistency checks); and within a broader context, such as an entity's entire history. Any violation of these specifications of validity are treated as errors.

5.2 Error Versus Invalid Data

Error data may be contrasted with good data--errors are mistakes by a user. It is usual for an information system to process both error and good data. Hence, both error data and good data are valid for a program component that handles error processing. Invalid data, then, is data that is unspecified for a program component. The distinction between error and invalid data is depicted below:



Since programs must be designed to process error data as well as good data, the data structures on which the program design is based must include errors as well as good data.

5.3 Error Processing Design Objectives

The design of data structures which accommodate errors is not easy. Many structures are possible, and we must make choices that affect the kind of errors that can be distinguished, and affect how good data is processed in the presence of errors. These choices are properly the concern of cognitive scientists and man-machine interface specialists; the choices are influenced by the types of errors that are likely to occur and how easily the user can correct them.

As programmers, we may set ourselves the following design objectives:

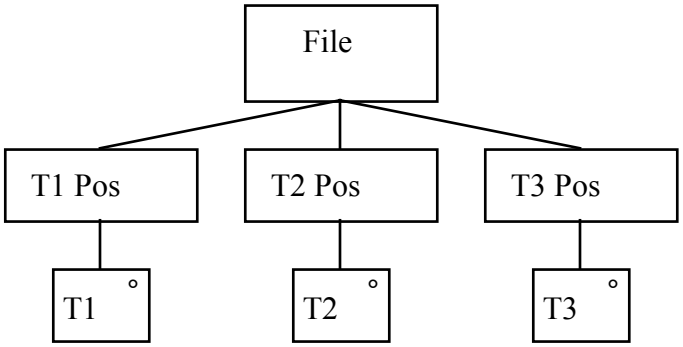
(i) The data structure should be known to the user, since it reflects the model of the problem; included, is the part of the data structure that deals with errors.

(ii) Diagnosis of errors should reflect the data structure - in fact, the diagnostic is directly related to a named component of the data structure. The user should receive a clear interpretation placed on his or her data.

(iii) Insofar as possible, the data structure should be designed so as not to interfere with the processing of good data.

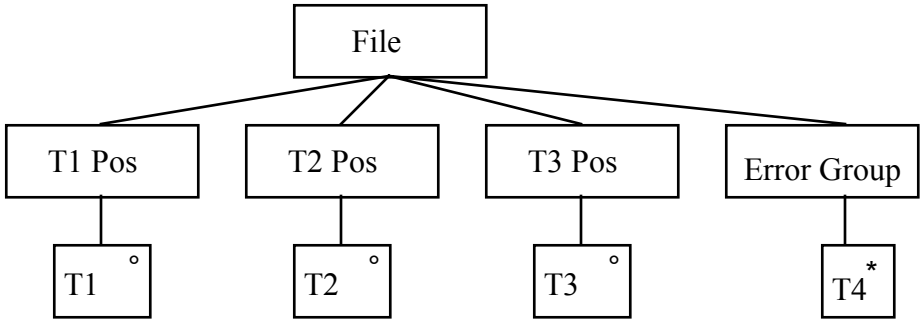
Let us illustrate these principles of error processing design with two examples.

(1) A transaction file contains up to three transactions, identified as T1, T2 or T3;. The file may be empty or contain any combination of the three transactions; but if present, T2 must precede T3 and if T1 is present it must precede T2. We can depict the file structure as below:



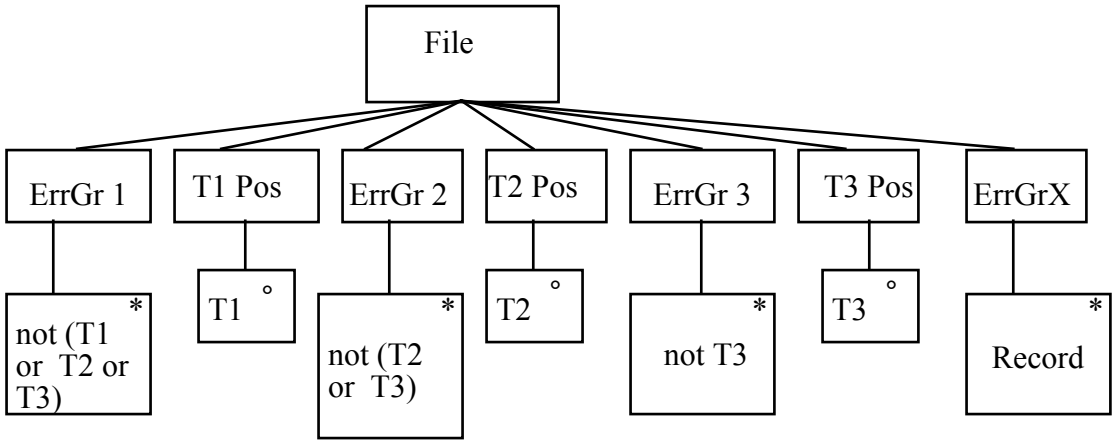
The diagram reflects the specification that any or none of T1, T2, or T3 may be present, and if present, they must be present in the sequence T1 before T2 before T3.

We can easily devise a structure to account for all possible errors such as the following:



Suppose our transaction file consists of TX, T1, T2, T3. Since TX is neither a T1, a T2, or a T3, it will fall into the error group together with its successors. This data structure has the unfortunate effect that any error causes the remainder of the file to be treated as an error.

The third design goal, to design the data structure so as not to interfere with the processing of good data, is difficult to achieve. Instead of the data structure shown above, we need to use some structure as:

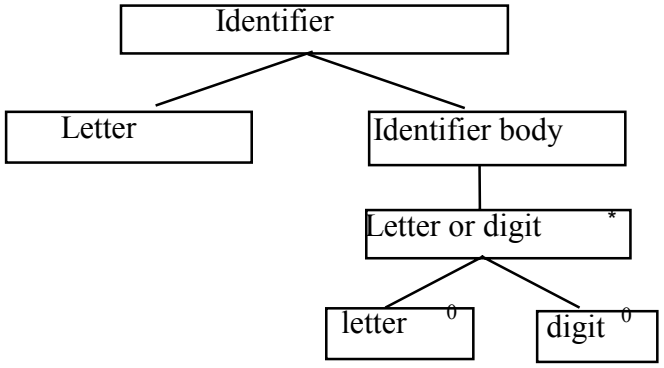


This structure is simple, but cumbersome and repetitive. Structures which take account of errors are often so.

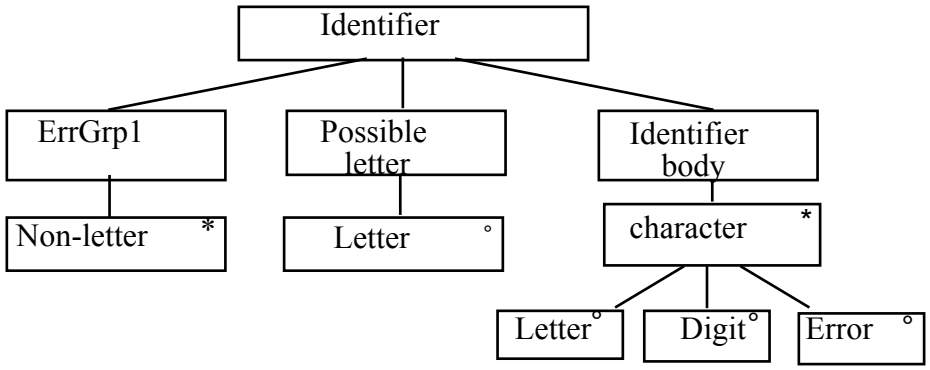
(2) An identifier in Pascal consists of a letter (a-z) followed by any number of letters or digits. Some examples of valid identifiers are: A, Alpha, A1k. We wish to detect and print out a diagnostic for two error types:

1A identifier doesn't begin with a letter
A1+2 identifier contains illegal character(s)

The structure diagram for well-formed identifiers is given below:



When we consider the errors that the program must be designed to process together with well-formed identifiers, we arrive at the following structure diagram that defines valid data for the program:



To design the program, we follow the usual JSP design steps. The diagnostic messages should be understandable and reflect the data structure on which the program is based. For example, in response to the identifier input

1AL+PH-A

the program might display the following:

Character	Message
1	Identifier doesn't begin with a letter
+	Illegal character in identifier body
-	Illegal character in identifier body

Just as the syntax diagrams for Pascal syntactic constructs are available to the user of Pascal, so, too, the data structure on which the compiler's syntax analyzer is based should also be available. Many compilers prior to the 1980's omitted a description of how syntax errors were handled, and may explain their sometimes arbitrary and user-unfriendly treatment of errors.

5.4 Valid and Invalid Data

Data is valid for a program component if the operation of the component is specified for the data; data is invalid for a program component if the operation of the component is unspecified for the data.

It follows that both good and error data are valid for a program component that is concerned with error processing.

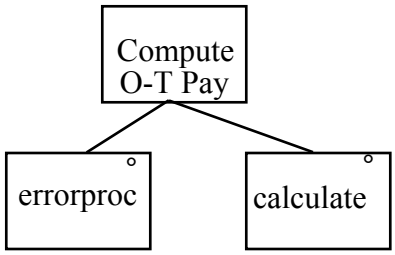
A program component that receives its input directly from a human input, or from an unreliable machine source, should be designed to accept all possible input data as valid, and hence its operation must be specified for all data.

However, a component that receives its input from another component of the same system must not check the validity of its own input, but must be able to rely on the correct operation of the component that invokes it. The attempt to have every component check its own input is self-contradictory, as the following example shows:

We wish to calculate overtime pay as part of a payroll system. Calculation is specified for times not exceeding 11 hours and 59 minutes. The result is a data item, overtime-pay. If we cannot rely on the values of hours and minutes in the input data, we may reasonably write:

```
if (hours > 11) or (minutes > 59)
then
   errorproc
else
     calculate (hours, minutes, overtime);
```

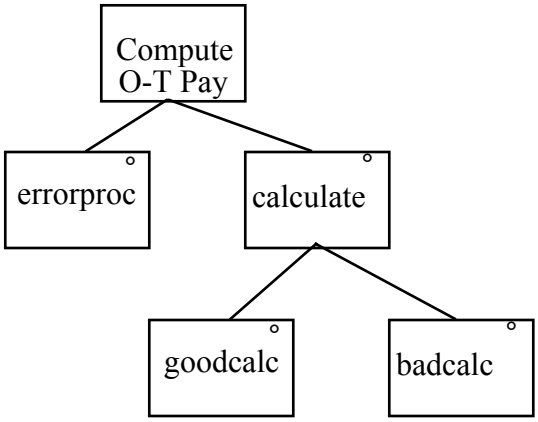
The structure of this component is:



The procedure, calculate, is relying on the correct functioning of the component, Compute O-T Pay, to ensure the validity of the hours and minutes values. Otherwise, the procedure calculate must itself check its inputs with a statement like:

```
if hours > 11 or minutes > 59
then
     badcalc
else
     goodcalc;
```

This gives as the complete structure:



If we apply the same reasoning to goodcalc, and to each of its "good" successor components, we are driven to an infinite regress! We must reconcile ourselves to the idea that the program component calculate will be unspecified for certain data inputs.

We can summarize our conclusions about designing components with respect to valid data as follows:

(i) Every component specification must define precisely what data is valid for the component;

(ii) Every component must be designed and coded on the assumption that its data is valid;

(iii) If component B is a part of component A, then A is responsible for ensuring that the data passed to B is valid for B.

Exercises

(i) Using the syntax diagrams as the definition of good data, design data structures to process possible user errors for the following Pascal syntactic entities: (a) integer; (b) real; (c) **program** heading; (d) **const** declaration

6. Recognition Difficulties and Backtracking

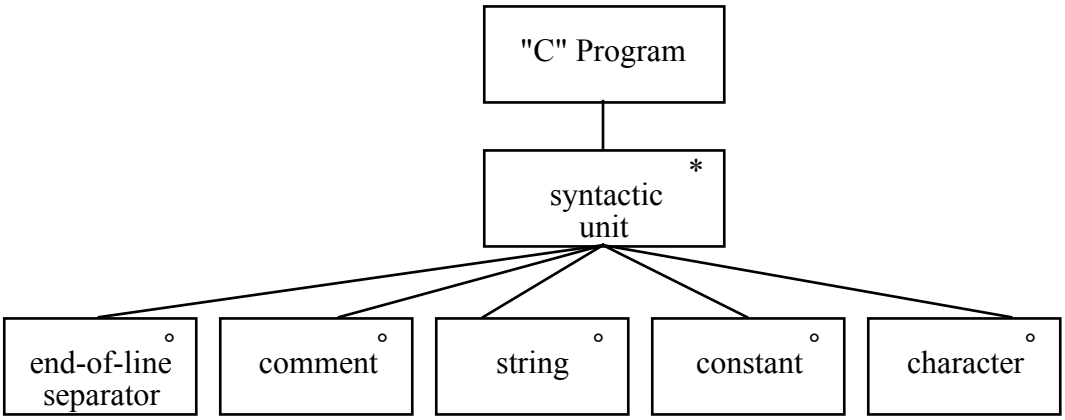
6.1 Multiple read-ahead rule

Sometimes a condition to be evaluated requires us to look ahead beyond the current character or record. Consider the following example:

> Write a program to remove all comments from a C program. A comment in C begins with the characters, /*, and ends with the characters, */. Take note of the fact that C allows strings, any characters enclosed by double-quotes ("), as well as constants, which are enclosed in single quotes.[8]

Although the input and output files are both text files, we will avoid explicitly modeling the input file as an iteration of lines for reason that will become evident in the later chapter dealing with structure clashes. It will suffice on reading the input file to recognize an end-of-line separator so that we can write an end-of-line separator, thus preserving the input line structure on the output file. In addition, we need to recognize a comment, a string, a constant, and a character.

The structure diagram of the input file is:



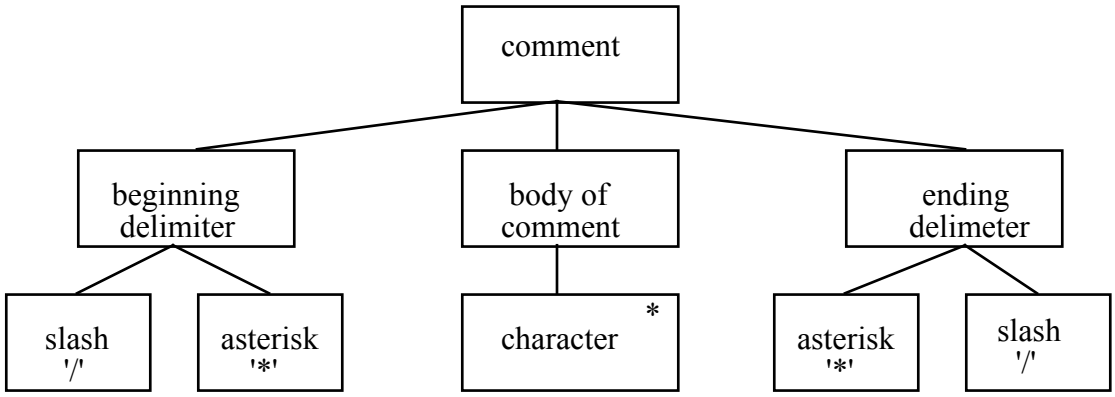
The output file has the same structure without comments.

In order to recognize a "C" comment, we need to read two consecutive characters. A dynamic view of the situation is to read the first character, and then, depending on whether we have a slash character, to read the second character, and, if it is an asterisk, to record the fact that we have a comment, if we are not processing a string. We may be led to use a Boolean variable to store when we are processing a string or constant.

This dynamic view hides the underlying structure of a comment shown below:

---

[8] The example is from "The C Programming Language" by Brian Kernighan and Dennis Ritchie, Second Edition, Exercise 1-23.



An approach based on the static view of a comment's data structure is to examine two consecutive characters for the two-character pattern, '/*'. Recognition of a comment requires that we read two characters prior to the selection.

In the situation where we have to perform several read operations to evaluate a condition, we need a different rule for allocating the read operations. It is called the multiple read-ahead rule:

Multiple read-ahead rule:
(1) determine n--the number of characters or records to be read ahead
(2) declare n record areas, each with a data part and an eof flag
(3) define the Nread procedure
(4) allocate n Nread operations at the beginning, immediately after the open operation, and 1 Nread operation at the end of the component that processes a record (character).

The structure text and data structure to implement the multiple read ahead rule are shown below:

(a) structure text (b) data structure

```
Nread    seq
 area0 := area1;              |eof0  | data0|   |rec0|
 area1 := area2;              |eof1  | data1|   |rec1|
          ...                                    ...
 areaN-1 := areaN;            |eofN | dataN|  |recN|
 eofinfile := eof(infile);
     Nnew   select <not eofinfile>
          readinfile(infile, areaN);
     Nnew  end
Nread  end
```

Here is a Pascal implementation of the Nread procedure:

```
type
    area = record
            c: char;
            eofbit, eolnbit: boolean;
        end;
var
    f: text;
    area0, area1: area;
procedure xread;       {read a character from a text file}
begin
    area0 := area1;
    area1.eofbit := eof(f);
    with area1 do
        if not eofbit then
            begin
                eolnbit := eoln(f);
                c := f^;
                get(f);
            end;
end;
```

Nread uses the record area rec0 as the current record, and the records in rec1, rec2, ..., recn only to look ahead n characters or records. A more efficient procedure would point to
the record areas instead of moving data from one to another.

Here is the main program for this example:

```
program stripcomments (input, output);
{ program strips comments from a "c" program" }
{  Kernighan&Ritchie  p. 34, exercise 1-23 }
const
    slash = '/';
    asterisk = '*';
    strdel = '"';           {string delimeter}
    constdel = "'";         {constant delimeter}
    infilename = 'data place 52:Development:JSP.pas:cprogram';
begin
    reset(f, infilename);
    xread;
    xread;        {2 read-aheads}
    while not area0.eofbit do
        if (area0.c = slash) and (area1.c = asterisk) then
{process comment}
            begin
                xread;   {strip beginning slash}
```

```
                xread;   {strip beginning asterisk}
                while not ((area0.c = asterisk) and (area1.c = slash)) do
                    xread;  {strip character of comment}
                xread;  {strip ending asterisk}
                xread;  {strip ending comment}
            end
        else if (area0.c = strdel) then
{process string}
            begin
                write(area0.c);
              {write beginning delimiter}
                xread;
                while (area0.c <> strdel) do
              {process body of string}
                    begin
                        write(area0.c);
                        xread;
                    end;
                write(area0.c);
              {write ending delimiter}
                xread;
            end
        else if (area0.c = constdel) then
{process constant}
            begin
                write(area0.c);
              {write beginning delimiter}
                xread;
                while (area0.c <> constdel) do
              {process body of constant}
                    begin
                        write(area0.c);
                        xread;
                    end;
                write(area0.c);
              {write ending delimiter}
                xread;
            end
        else if area0.eolnbit then
            begin
                writeln;
                xread
            end
        else
{process anything else}
            begin
```

```
                write(area0.c);
                xread;
            end;
    close(f);
end.
```

Note: Compare the same program written in C, most of which is shown below. Note in particular that the multiple read-ahead rule is followed:

```
#include <stdio.h>
#define SLASH       '/'
#define ASTERISK    '*'
#define DQUOTE      ""
#define SQUOTE      '\x27'

char c,d;    /*global variables */

void nextchar(void);  /* multiple read-ahead; the first character is readinto c,
                         the second into d */
main()
{
nextchar();     /* multiple read ahead at start */
nextchar();
while (c != EOF)
    if (c == SLASH & d == ASTERISK)  {   /* process comment */
        nextchar();  /* strip beginning slash */
        nextchar();  /* strip beginning asterisk */
        while (! (c == ASTERISK & d == SLASH))
            nextchar();  /* strip character of comment */
        nextchar();  /* strip ending asterisk */
        nextchar();  /* strip ending slash */
        }
    else if (c == DQUOTE)  {  /* process string */
        putchar(c)  /* output beginning delimeter */
        nextchar();
        while (c != DQUOTE)  {  /* process body of string */
            putchar(c);
            nextchar();
            }
        putchar(c)  /* output ending delimeter */
        nextchar();
        }
    else if (c == SQUOTE)  {  /* process constant */
        ...
        ...
        }
```

```
    else {  /* process anything else */
        putchar(c);
        nextchar();
        }
return 0;
}   /* end main */

void nextchar(void)
{
    c = d;
    if (c != EOF)
        d = getchar();
}
```

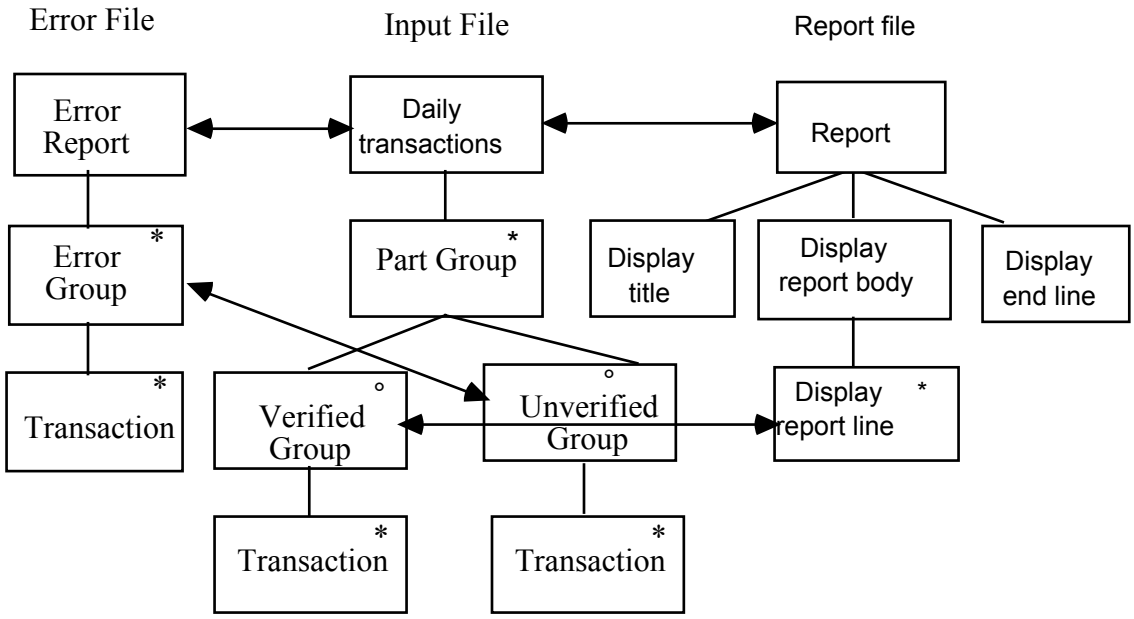
6.2 Backtracking

The multiple read-ahead technique discussed in the previous section is fine as long as we know that we can evaluate a condition after a known number of records or characters. But this is not always the case. Sometimes we cannot recognize a condition after any fixed number of reads. In this situation we need a different technique.

Let us return to our earlier example which computed the total net movement for each part group in a file of warehouse transaction records sorted by part number. As a quality control measure, each shipment and order must be verified by a warehouse supervisor when the goods are actually received into or issued out of the warehouse, and the supervisor's initials (a two-character code) are then recorded on each transaction. At the end of the day, it is desired to produce a report showing net movement for each part; however, parts are to be included on the report only if every shipment and order transaction has been properly verified, as indicated by a code in the initials field on each transaction record. Parts with one or more unverified shipment or order transactions, indicated by a blank in the initials field, should be written on an error listing; they are not to be included in the report (no total line should be generated).

We don't know how many shipment and order transactions exist for any part number. Thus, we don't know how many records we have to read in order to recognize whether we have a part all of whose shipments and orders have been verified.

Here are the data structures for the input and output files:



Each part group is either completely verified or not. There are innumerable ways that a part group may be unverified, and this creates a difficult recognition problem. If, however, we treat the selection as being ordered and place the verified group as the left component of the selection, then we don't have to recognize all of the ways of obtaining an unverified group--a group is unverified if it is not completely verified.

The design of our program proceeds in three stages:

(1) Design the program as if a friendly demon will tell us whether a group is verified or not. That is, at first, ignore the recognition difficulty and design the program as if the selection condition can be evaluated at the start of the selection.

Using JSP, we obtain the program structure text below:

```
CFILE-PREPORTS  seq
    reset(transfile);
    rewrite(reportfile);
    rewrite(errrlist);
    xread(transfile);    {read header}
    P-REPORTBODY  iter until eofbit
        groupkey := trans.key;
        GROUP-OUT  select verified-group
            total := 0;
            xread(transfile);
            VERIFIED-GROUP  iter until eofbit or trans.key <> groupkey
                PROCESS-TRANS  select  trans.type = 'S'
                    total := total + amount;
                PROCESS-TRANS  alt  trans.type = 'O'
```

```
                total := total - amount;
            PROCESS-TRANS  end
            xread(transfile);
        VERIFIED-GROUP  end
        writeln(reportfile,groupkey, total);
    GROUP-OUT  alt  unverified-group
        UNVERIFIED-GROUP   iter until eofbit or trans.key <> groupkey
            writeln(errlist, trans.rec);
            xread(transfile);
        UNVERIFIED-GROUP  end
    GROUP-OUT  end
P-REPORTBODY  end
close(transfile);
CFILE-PREPORTS  end
```

We cannot, of course, translate the structure text into a program, since we have no way of specifying the condition for a verified group. However, we could, acting as our own friendly and omnipotent demon, code the first transaction of each part group with a code indicating whether every transaction in the group had been verified or not, and thus informally demonstrate the program's acceptability.

(2) In the second stage of program design, instead of selecting a verified part group, let's posit that we have a verified group. If we find an unverified transaction in the group, we will quit this component, and go to the component that processes an unverified group. Since this control structure is not currently implemented in programming languages, we will need to introduce it. The structure text and pseudocode for this backtracking control structure are shown below:

```
structure text                  Pascal implementation

A   posit                   begin  {posit}
    do B;                             do B;
A   quit <condition>              if <condition> then
    do C;                                 goto 1; {quit, go to admit part}
A   admit                         do C;
    do D;                         goto 2;   {finished--no error}
A   end                       1:  {admit error}
                                  do D;
                              2:  end
```

Note that if we quit the posit component, control is passed to the start of the admit component; while if we do not quit the posit component, control is transferred, after we have completed any statements in the posit component that follow the quit statement, to the end line of the posit..admit..end control structure. Because the control structure has not been implemented, we must resort to using the forbidden goto statement. Note, however, that this is not a case of an unrestricted goto; the labels are synonyms for the

admit and end lines of a semantically well-defined but unimplemented control structure. Moreover, it is at the heart of our method to design intelligible programs--programs that reflect the problem structure. Designing our program to accommodate the rule "don't use goto statements" instead of the problem will result in a program structure that is distorted and will be more costly to maintain.

The program structure text reflecting the second design stage with the posit..admit control structure is shown below:

```
CFILE-PREPORTS  seq
    reset(transfile);
    rewrite(reportfile);
    rewrite(errrlist);
    xread(transfile);    {read header}
    P-REPORTBODY  iter until eofbit
        groupkey := trans.key;
        GROUP-OUT  posit verified-group
            total := 0;
            xread(transfile);
            VERIFIED-GROUP  iter until eofbit or trans.key <> groupkey
                quit  VERIFIED-GROUP if trans.init = blanks
                PROCESS-TRANS  select  trans.type = 'S'
                    total := total + amount;
                PROCESS-TRANS  alt  trans.type = 'O'
                    total := total - amount;
                PROCESS-TRANS  end
                xread(transfile);
            VERIFIED-GROUP  end
            writeln(reportfile,groupkey, total);
        GROUP-OUT  admit unverified-group
            UNVERIFIED-GROUP  iter until eofbit or trans.key <> groupkey
                writeln(errlist, trans.rec);
                xread(transfile);
            UNVERIFIED-GROUP  end
        GROUP-OUT  end
    P-REPORTBODY  end
    close(transfile);
CFILE-PREPORTS  end
```

(3) In the third and last stage of design, we identify and deal with side effects. Side effects may be categorized into three types:

(i) neutral - changes in the state of the computation resulting from the execution of the posit block have no adverse affect on the final result;

(ii) positive - computations resulting from the posit are valid in the admit part, and save the trouble of recomputing;

(iii) negative - changes in the posit have to be undone; the state of the computation has to be restored at the start of the admit block to the state that existed at the start of the posit block

It is from third type of side effects that the name "backtracking" comes: we need to "undo" what we've done from the beginning of the posit until we discover that the posit condition does not hold, that is, "backtrack" or, in other words, restore the state of the computation at the start of the posit part. Restoring the state of the computation that existed at the start of the posit suggests that we need to save the state of the computation at this point.

In our problem we have negative side effects that are associated with reading the input file. Adverse side effects are usually associated with reading/writing of serial files. The best and easiest way to deal with this side effect is with procedures that 'note' and 'restore' the file. The note procedure is invoked immediately after the posit operation, and the restore procedure immediately after the admit statement.

The complete PASCAL program with the note and restore procedures follows:

```
   program backtracking (input, output);
{example adapted from "Constructive Methods of Program design" }
{    by M. Jackson to illustrate backtracking}
   const
      fname = 'data place 52:Development:JSP.pas:WarehouseTrans';
      reportname = 'data place 52:Development:JSP.pas:Report';
      errorname = 'data place 52:Development:JSP.pas:ErrorList';
      blank = '   ';
   var
      f, report, error: text;
      sum, groupkey, cnt, savcnt: integer;
      eofbit: boolean;
      key, qty: integer;
      initials: packed array[1..3] of char;
   label
      1, 2;

   procedure xread;
   begin
      eofbit := eof(f);
      if not eofbit then
         readln(f, key, qty, initials);
   end;

   procedure note;
   begin
      savcnt := cnt;
```

```
    end;

    procedure restore;
    begin
        reset(f);
        cnt := 0;
        repeat
            xread;
            cnt := cnt + 1;
        until cnt = savcnt;
        groupkey := key;   {error group}
    end;

begin  {main program}
    reset(f, fname);
    rewrite(report, reportname);
    rewrite(error, errorname);
    cnt := 0;
    xread;
    cnt := cnt + 1;
    writeln(report, ' Daily Net Movement Report ...');
    writeln(error, ' Error List');
    while not eofbit do
        begin
            groupkey := key;
    {posit verified group}
            note;
            sum := 0;
            while not eofbit and (groupkey = key) do
                begin
                    if (initials = blank) then
                        goto 1; {quit verified group}
                    sum := sum + qty;
                    xread;
                    cnt := cnt + 1;
                end;
            writeln(report, '  ', groupkey, '   ', sum);
            goto 2;
    {end verified group}
1:  {error group}
            restore;
            while not eof(f) and (groupkey = key) do
                begin
                    writeln(error, ' ', key, qty, initials);
                    xread;
                    cnt := cnt + 1;
```

```
                    end;
        {end error group}
2:      end; {while not eof(f)}
    close(f);
end.
```

6.3 Backtracking (within iteration)

6.3.1 **quit** in iteration

Consider the following problem:

A file consists of a sequence of integers. Print the integers as long as their cumulative sum doesn't exceed 100.

We would probably structure the program text as follows:

```
F-SEQ  seq
    reset(f);
    xread(f,x);
    sum := 0;
    F-BODY  iter while not eofbit
        sum := sum + x;
        F-WRITE  sel  <sum <= 100>
            write(x);
        F-WRITE  end
        xread(f,x)
    F-BODY   end
F-SEQ  end
```

Note that the problem states that we are done when the cumulative sum exceeds 100. In our structure text, however, we continue iterating until the end of file is reached.

We will introduce **quit** as a new element in our structure text. Its syntax is

quit <component> <condition>

and has the meaning "quit from the specified program component if the condition specified is true."

Our program structure text becomes:

```
F-SEQ  seq
    reset(f);
    xread(f,x);
    sum := 0;
```

```
    F-BODY  iter  while not eofbit
        sum := sum + x;
        quit  F-BODY if sum >100;
        write(x);
        xread(f,x);
    F-BODY  end
F-SEQ  end
```

We could have achieved the same result by putting a compound condition at the head of the iteration as shown below:

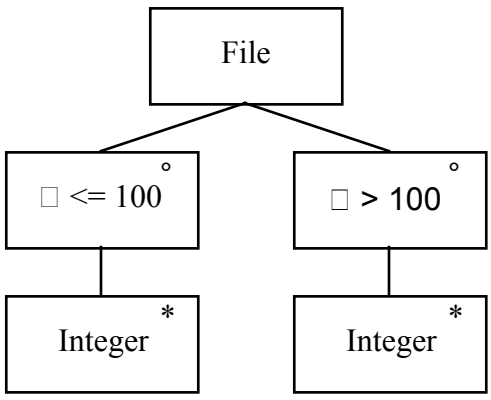
```
F-SEQ  seq
    reset(f);
    xread(f,x);
    sum := 0;
    F-BODY  while not eofbit and (sum <= 100)
        sum := sum + x;
        write(x);
        xread(f,x);
    F-BODY  end
F-SEQ  end
```

We don't really need a backtracking construct here, because there are no adverse side-effects.

6.3.2 Backtracking in iteration

Let us consider a slight modification to the problem presented above. The program is to print the entire file only if the cumulative sum of the integers exceeds 100; in any case, the cumulative sum is to be printed.

The input file structure is as follows:



The design of the output and basic program structures is left as an exercise.

We will treat this problem as one requiring backtracking and proceed in three stages as before:

Stage 1: Design the program as a selection

```
F-SEQ  seq
    reset(f);
    xread(f,x);
    sum := 0;
    F-BODY  sel   {sum <= 100}
        F-BODY-ITER1  while not eofbit
            sum := sum + x
            xread(f,x);
        F-BODY-ITER1  end
    F-BODY  alt   {sum > 100}
        F-BODY-ITER2  while not eofbit
            sum := sum + x;
            write(x);
            xread(f,x);
        F-BODY-ITER2  end
    F-BODY  end
    write(sum);
F-SEQ  end
```

Stage 2: Change the selection to a **posit**...**admit**

In addition to the lines with **sel** and **alt**, we need to add

quit F-BODY **if** sum >100;

immediately after the statement that updates the value of the variable, sum, in the component, F-BODY-ITER1.

Stage 3: Deal with side effects

When we find that the cumulative sum exceeds 100 in the posit component, we have already read part of the input file without having printed it. There is more than one way to deal with this side effect. One way is to reset the input and initialize the sum at the start of the admit--in effect, start over. We obtain the structure text shown below:

```
F-SEQ  seq
    reset(f);
    xread(f,x);
    sum := 0;
    F-BODY  posit  {sum <= 100}
```

```
        F-BODY-ITER  while not eofbit
            sum := sum + x
            quit  F-BODY if sum >100;
            xread(f,x);
        F-BODY-ITER1  end
    F-BODY  admit   {sum > 100}
        reset(f);               {reinitialize}
        xread(f, x);
        sum := 0;
        F-BODY-ITER2  while not eofbit
            sum := sum + x;
            write(x);
            xread(f,x);
        F-BODY-ITER2  end
    F-BODY  end
    write(sum);
F-SEQ  end
```

Another way to deal with the undesirable side effect of having read some of the input file without printing when the posit condition has been disproved is to save the integers read to a print file; if and when the cumulative sum exceeds 100, then, at the start of the admit component, we will copy the print file to the printer. The partial sum that has been computed in the posit component is mostly a positive side effect--we must only undo the last operation that caused the sum to exceed 100; the rest of the computation of the cumulative sum does not have to be recomputed. The structure text is shown below:

```
F-SEQ  seq
    reset(f);
    xread(f,x);
    sum := 0;
    F-BODY  posit  {sum <= 100}
        rewrite(g);           {print file}
        F-BODY-ITER1  while not eofbit
            sum := sum + x
            quit  F-BODY if sum >100;
            write(g, x);
            xread(f,x);
        F-BODY-ITER1  end
    F-BODY  admit   {sum > 100}
        sum := sum - x;         {undo sum := sum + x}
        rewrite(g);
        gread(g, x);            {gread is file procedure for g}
        F-PRINT  iter while not eofgbit       {eofgbit associated with gread}
            gread(g, x);
            write(x);
```

```
        F-PRINT  end
        F-BODY-ITER2  while not eofbit
            sum := sum + x;
            write(x);
            xread(f,x);
        F-BODY-ITER2  end
    F-BODY  end
    write(sum);
F-SEQ  end
```

Finally, it is interesting to note what happens if we change the order of selection--that is, we posit that the cumulative sum exceeds 100. In this case, we must deal with the side effect of having already printed part of the file if it turns out that cumulative sum does not exceed 100. To deal with this side effect, we will, instead of writing directly to a printer, write to a print file as we did previously. Then, if only if the cumulative sum of the entire file exceeds 100, we will print the print file. The structure text is shown below:

```
F-SEQ  seq
    reset(f);
    xread(f,x);
    sum := 0;
    F-BODY  posit  {sum > 100}
        rewrite(g);      {g is print file}
        F-BODY-ITER1  while not eofbit
            sum := sum + x;
            write(g,x);
            xread(f,x);
        F-BODY-ITER1  end
        quit F-BODY if sum <= 100;
        rewrite(g);      {now get set to write the print file to the printer}
        xread(g, x);
        F-PRINT   iter while not eofbit
            write(x);
            xread(g, x);
        F-PRINT  end
    F-BODY  end
    write(sum);
F-SEQ  end
```

Note that we have computed the sum correctly in the posit component--a positive side effect. In fact, all of the work of the admit component has been done in the posit component, and we can discard it entirely.

Exercises

(i) The data structure given for the multiple read-ahead rule is inefficient in that it requires that (n-1) record areas be copied prior to reading into the current record area. Design a more efficient data structure and the associated read procedure.

(ii) Write the program to produce the verified net movement report and unverified error listing without using the go to statement. Is the structure intelligible (does it model the problem structure)?

(iii) Design a program to print out the contents of a file if it contains more than 50 records.

(iv) A file consists of alternating sequences of integers. Each sequence begins with an identifier containing a sequence number and the number of integers in the sequence. A program is to be written to copy the sequences as a single sequence. The program should terminate when the cumulative sum of all of the integers in sequences read so far exceeds a parameter, n.

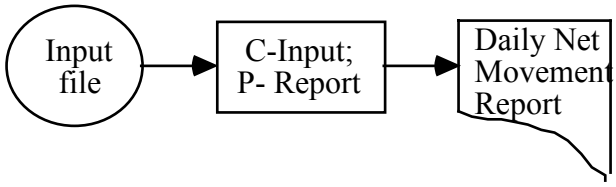
7. Structure Clashes and Program Inversion

7.1 Structure Clashes

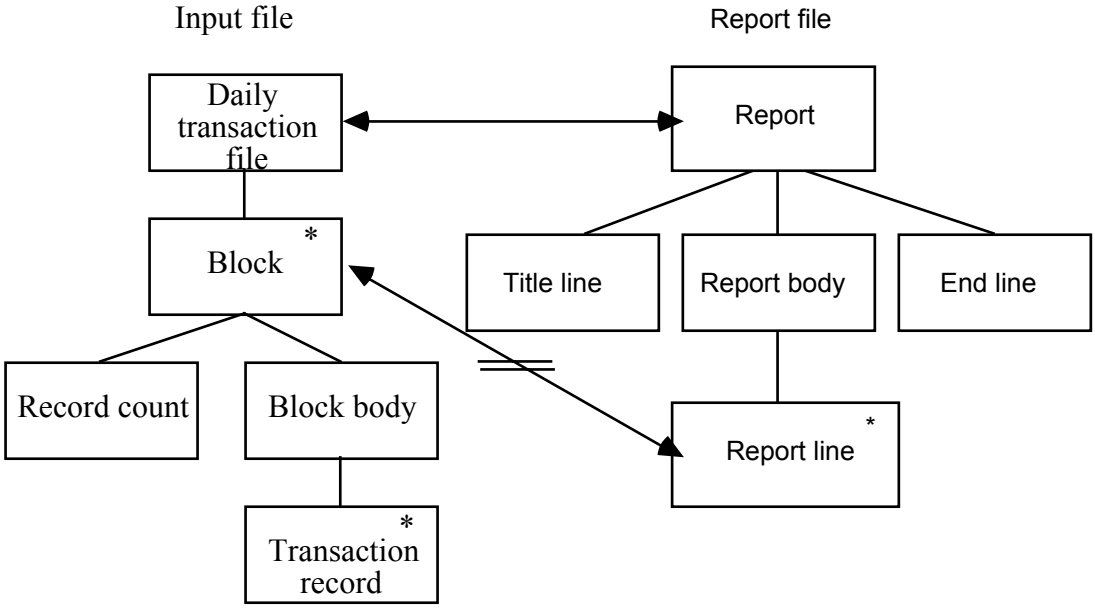
The design method that has been presented in the preceding chapters works when there is a correspondence between input and output data structures. What happens when such correspondences don't exist, as in the following example?

> Consider the input file of the net movement problem in section 4.1.2. The file consists of daily transactions sorted by part number; each part number may have one or more transactions--either a receipt into the warehouse or an order out of the warehouse. Each transaction contains a transaction code, a part-identifier, and a quantity received or ordered. A program is to be written that prints a line for each part number showing the net daily movement for that part number into or out of the warehouse. Suppose the input file is blocked, with each block containing a record count followed by a number of records.

The system diagram is shown below:



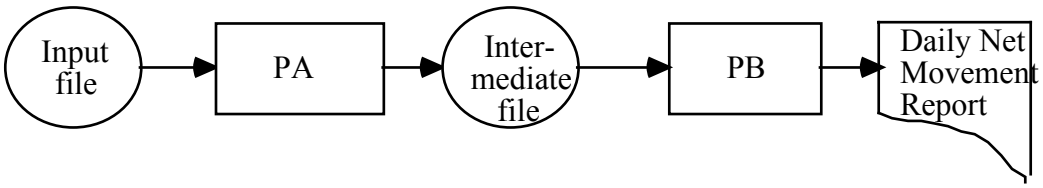
The structure of the input and output files is shown below:



The report file structure is the same as was shown previously in section 4.1.2; however, the input file structure shows the arrangement of records into blocks, but not the arrangement of records into groups--we cannot depict the structure of blocks and groups in a single diagram.

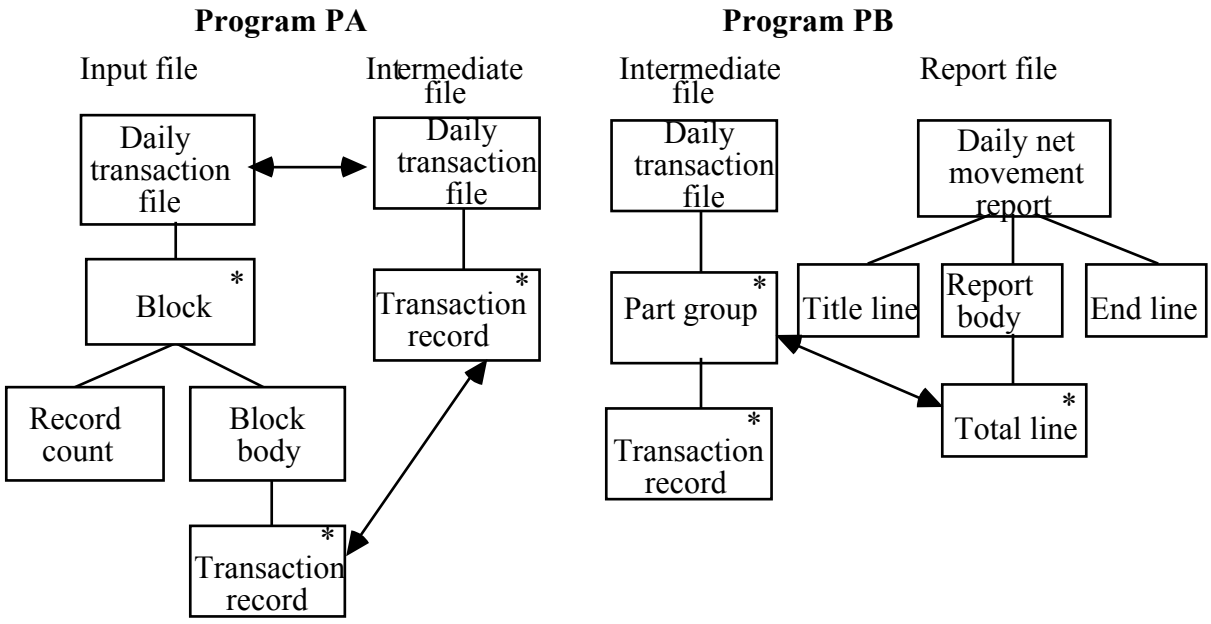
A report line does not correspond to a block. The input and output structures are thus not compatible. The essence of our difficulty is that the program must have an operation that is executed once per block and an operation that is executed once per group, so there must be both a block component and a group component; but we cannot have a single program structure with process block and process group components. We have a boundary clash--the boundaries of blocks are not synchronized with the boundaries of groups.

The solution is to decompose the program into two simple programs, as shown below:



Program PA consumes the input file of blocks of records and produces an unblocked file of transactions. Program PB consumes the groups of unblocked transaction records and produces the required report.

The input, intermediate and report files, together with their correspondences, are shown below:



Programs PA and PB can both be written using JSP: the input and output files of PA have correspondence at the top and record levels; a net movement total line in the report produced by PB corresponds to a set of transaction records corresponding to a part group in the input file. The structure text for PA and PB is shown below:

```
PA  seq
  reset(infile);
  xread(infile, block);
  rewrite(outfile);
  PA-BLOCK  iter <while not eofbit>
    j := 1;
    PA-DEBLOCK  iter
        <while not (j>reccnt)>
      outrec.itemno := inrec.itemno[j];
      outrec.transcode := inrec.transcode[j];
      outrec.qty := inrec.qty[j];
      write(outfile, outrec);
      j := j + 1;
    PA-DEBLOCK  end
    xread(infile,block);
  PA-BLOCK  end
  close(infile);
  close(outfile);
PA   end
```

```
PB  seq
  reset(infile);
  xread(infile, rec);
  writeln(' Daily Net Movement Summary ');
  writeln;
  PB-REPORTBODY  iter <while not eofbit>
    groupid := rec.itemno;
    netqty := 0;
    PB-GROUP  iter  <while not eofbit
          and (groupid = rec.itemno)>
      netqty := netqty + rec.qty;
      xread(infile, rec);
    PB-GROUP  end
    writeln(' ', groupid, ' ', n   etqty);
  PB-REPORTBODY  end
  close(infile);
PB  end
```

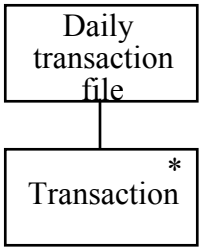
Decomposition of a complex program into two or more simple programs has the following advantage: The programs we obtain are distinct. A serial file forms a boundary between any pair of programs. We don't have to think "dynamically". For example, we don't need to ask, "What if a group extends over several blocks?" or "What if a group has no data records?" We know our programs are correct, because we can think in terms of static data structures. Simple programs are a satisfactory high-level design component--they are bigger than the control structures of structured programming; at the same time, they have more precise criteria than modules.

Our solution is inefficient, however. By introducing an intermediate file, we have roughly doubled the execution time (in comparison with a program that produced a report without an intermediate file). We will learn a little later how to optimize our design by a simple program transformation, program inversion.

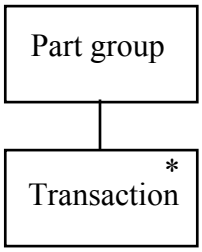
The "boundary" clash in this example is one type of structure clash. Another type of structure clash appears in the following example:

> Let us suppose that our input file is incompletely sorted by part number. Total lines for each part group on the report may be in any order.

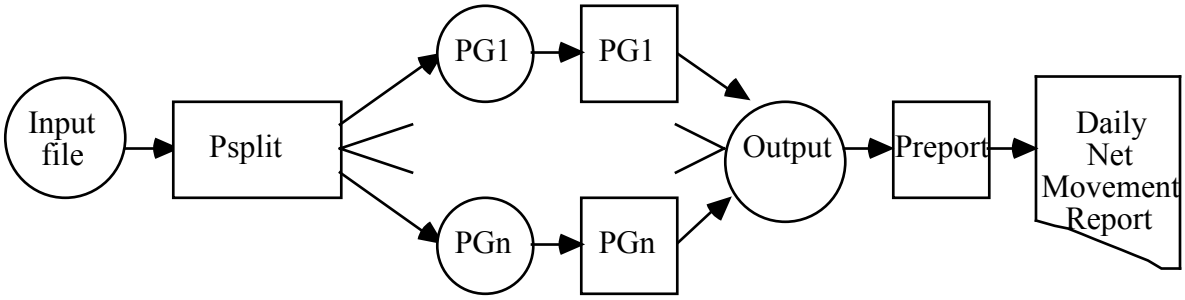
The input file has the following structure:



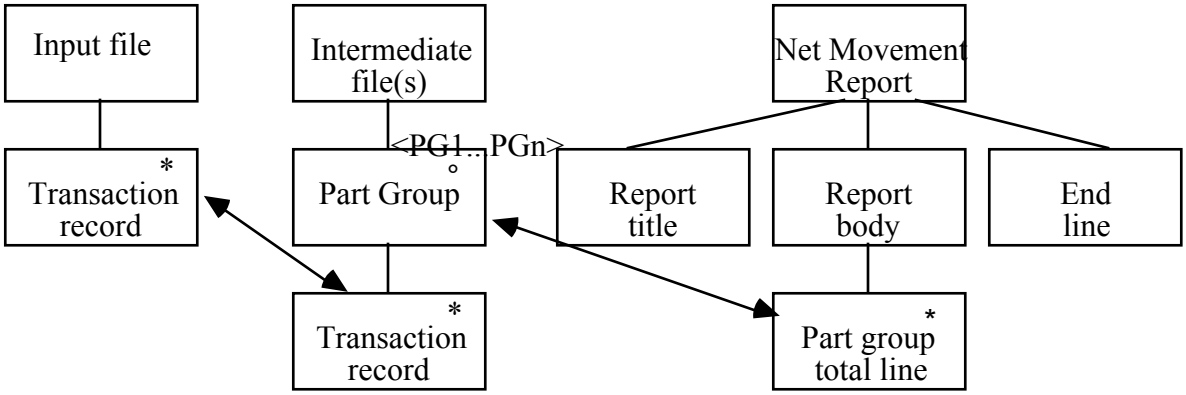
The input consists of groups, each of which has the from shown below:



Since the input file is not sorted completely by part number, we cannot show the group structure and the input file structure on one diagram. Our input file is an interleaving of part groups. To resolve this "interleaving" clash, we split the input file into part groups as shown below:



Each of the intermediate files, PG_1, PG_2, ...,PG_n has the structure of a part group. The programs, PG_1, PG_2,..., PG_n, that produce the Daily Net Movement Report each read a part group file and produce the report, each program contributing one total line to the report. The input and output data structures shown below for programs Psplit and Preport show correspondences and no structure clash; hence, we have no difficulty constructing the programs using the basic JSP design method:



7.2 Program Inversion

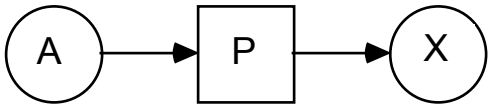
In the previous section, we examined how to deal with structure clashes, that is situations in which we cannot construct a program design because we don't find compatibility between input and output file structures. We looked at two types of structure clash, namely boundary clash and interleaving clash. We found that the solution to the structure clash in both cases is to decompose the program into two (or more) simple programs that can be designed using the steps of JSP. We noted that this method of decomposition has several advantages:

(1) We are able to use JSP, and can be certain that we have produced well-designed programs;
(2) The serial file produced by the first program provides a simple yet well-defined interface between the two programs.

However, we noted that this solution by decomposition is inefficient. We would like keep our design method, but be able to achieve an optimized solution.

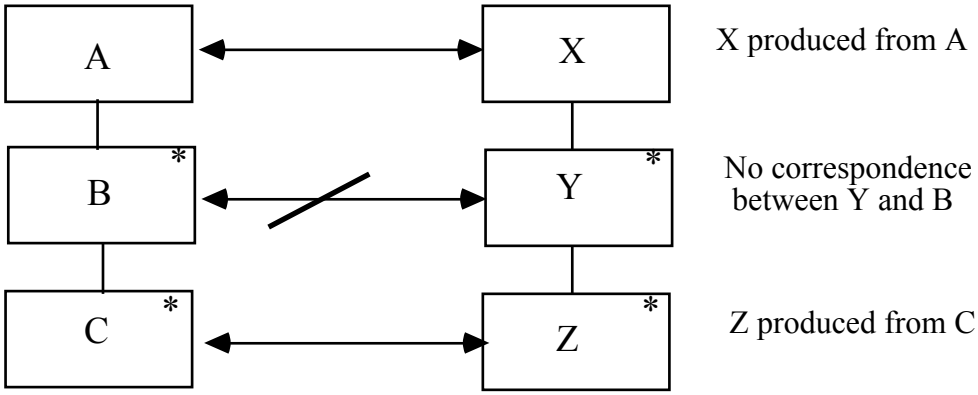
Program inversion achieves this objective, that is, we design simple programs using JSP and then invert one (or more) programs to optimize the design. Moreover, we gain this efficiency without introducing errors because program inversion is a well-defined (algorithmic) program transformation.

Let us reconsider the boundary clash problem in general terms. The system diagram is given in the following diagram:



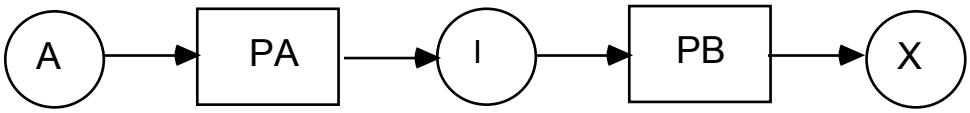
(A and X are serial data streams)

The structure of the serial files A and X is shown in the following diagram:

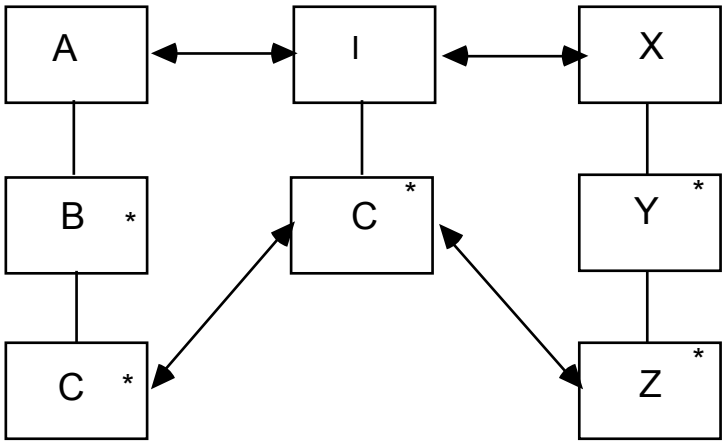


There is a 1-to-1 correspondence at the top level and at the lowest level, but there exists an intermediate level where no such correspondence exists.

The resolution of the structure clash is effected by decomposing the program P into two programs, PA and PB as shown below:



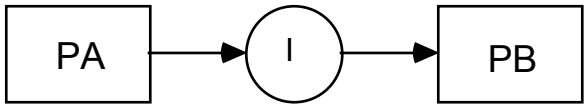
The data structures of the serial input file, A, the intermediate file, I, and the output file, X are shown below:



There is no structure clash between A and I, so the structure of PA can be constructed without difficulty; the same is true of I and X and PB. Whereas in general, the description of a file as an iteration of records is rarely sufficient--though always true--in designing a program, in the case of resolving a structure clash, this description is

invaluable. The intermediate file, I, contains the largest component common to both A and X--an idea analogous to the "greatest common divisor" of two integers.

What is important in the resolution of the structure clash is a decomposition that enables us to concentrate on a correct analysis of PA and PB. The processing for the system diagram

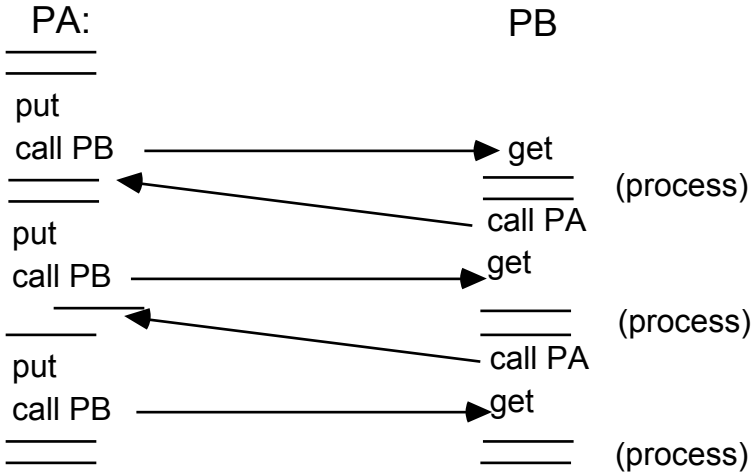


can be accomplished in a number of ways:

(1) Batch processing: PA produces the serial data stream, I, which is then processed by program PB.

Special case: If I can be stored in memory, there is no need for an external data stream, I, and P can be accomplished by executing in order subroutines PA and PB.

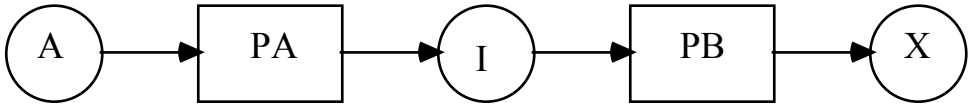
(2) Parallel processing: It would be more efficient if P2 processes each record of I as soon as it is created by PA instead of waiting until all of I has been produced. We can arrange PA and PB to be cooperating programs or coroutines. PA produces a record in a buffer and transfers control to PB which consumes (processes) the record and then transfers control back to PA again. The cooperation between PA and PB is depicted below:



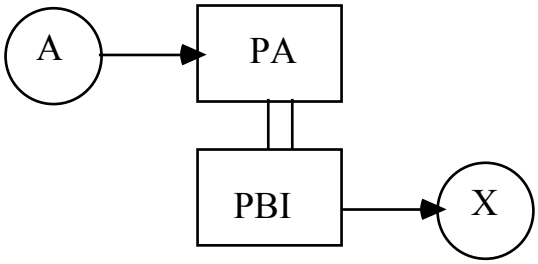
Alternatively, PA and PB can be written as independent tasks under control of a multi-programming task supervisor which manages the alternating suspend and resume between the two tasks.

(3) Quasi-parallel processing: program inversion

Multi-programming is expensive in system resources. We can achieve a solution of a structure clash more cheaply: instead of running PA and PB in parallel, we can convert one so that it runs as a subroutine of the other. We call the conversion process, program inversion. The system diagram:



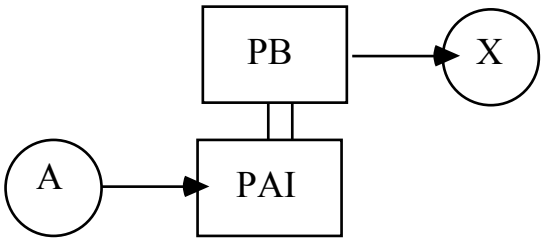
is rewritten as:



PA produces a record and invokes the subroutine PBI which uses it to produce X.

We say that PBI is inverted with respect to its input file.

or as:



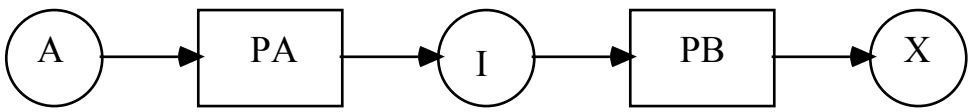
PB produces X, invoking the subroutine PAI to obtain the next record.

We say that PAI is inverted with respect to its output file.

Subroutines or procedures in FORTRAN, COBOL and Pascal (among others) do not store their return address in the calling program, and so the calling program does not know where to resume the subroutine. If a multiple ENTRY facility is used, the program, PA, is no longer independent of PB, since it must know which ENTRY point to invoke. We wish to design PA and PB independently of each other so that the **structure of each is based on the data structure representation of its own problem environment.**

Program inversion is a purely mechanical transformation of the independent programs, PA and PB, into a main program and subroutine, where the subroutine has a single ENTRY point and stores its return address within itself so that it is resumable.

Consider the boundary clash example again:



We can convert the program PA into a procedure PAI, which has the characteristics of an input procedure for I. Successive calls to PAI will 'open I', 'read I' (iteratively), and 'close I'. PAI passes the next record to PB and notifies PB when an end of file has been reached. (Alternatively, we could convert PB into a procedure PBI, which has the characteristics of an output procedure for I.) PA and PAI are **identical programs**. PAI is a resumable or variable-state procedure, that is, it performs some operations, suspends its execution (passes control back to PB), and when called again, resumes at the point where it left off. PAI must keep track of its state, and does so with a state vector (or activation record) that holds the values of any variables together with a text pointer to the next instruction in its text to be executed.

In our example, we know that the stream of operations that PAI must perform is: open, read, read,..., close.

Let us let PA represent our program to deblock our blocked warehouse transactions and PB the program to write our daily net movement summary report. Here is the structure text for PA followed by its transformation into the variable state subroutine, PAI:

```
PA  seq
   reset(infile);
   xread(infile, block);
   rewrite(outfile);
   PA-BLOCK  iter <while not eofbit>
     j := 1;
     PA-DEBLOCK  iter
           <while not (j >blockreccnt)>
        outrec.itemno := inrec.itemno[j];
        outrec.transcode := inrec.transcode[j];
        outrec.qty := inrec.qty[j];
        write(outfile, outrec);
        j := j + 1;
     PA-DEBLOCK  end
     xread(infile,block);
   PA-BLOCK  end
   close(infile);
   close(outfile);
PA   end
```

```
PAI  seq
     GOTO (5, 15) DEPENDING ON qs
5:   reset(infile);
     xread(infile, block);
     PA-BLOCK iter <while not eofbit>
       j := 1;
15:  PA-DEBLOCK iter
           <while not (j>blockrecnt)>
        outrec.itemno := inrecitemno[j];
        outrec.transcode:= inrec.transcode[j];
        outrec.qty := inrec.qty[j];
         j := j + 1;
         qs := qs + 1;
         return
       PA-DEBLOCK end
       xread(infile,block);
    PA-BLOCK  end
    close(infile);
    return
 PAI   end
```

The following changes transform PA to PAI:

(1) A GOTO <L1, L2, ..., Ln> DEPENDING ON qs is coded as the first statement of the text of PAI.

After each write statement in PA's text, the position in the text must be remembered explicitly. On the next invocation of PAI, the program must resume at that point. Since a procedure starts each invocation at the beginning of its text, there must be

a conditional jump from the beginning of the program text to the correct resume point. We have used the GOTO <L1, L2, ...> DEPENDING ON qs statement to express the resume mechanism. Here, L1, L2, ... , Ln represent label statements in PAI, and qs represents successive states of PAI. The variable, qs is assumed to be initialized to 0. The local variables of PAI together with a pointer to the point of resumption in the program text is called the state vector of PAI.

(2) The rewrite(outfile) in PA is deleted from PAI

(3) Instead of the write (outfile, outrec) statement in PA, PAI passes the next record to PB, increments qs to resume at the proper point of its text, and returns control to PB.

(4) When the end of file is reached, eofbit is set to true, and PAI passes it to PB.

(5) The close(outfile) statement in PA is deleted from PAI.

PA and PAI have the same structure. The transformation of the program PA to the variable state procedure PAI is straightforward, and can be accomplished algorithmically.
Because we have inverted the program PA with respect to the file I, we have optimized our solution--the file I no longer exists.

Here is the original program PB followed by the transformed program PB':

```
PB  seq
   reset(infile);
   xread(infile, rec);
   writeln(' Daily Net Movement Summary ');
   writeln;
   PB-REPORTBODY  iter <while noteofbit>
      groupid := rec.itemno;
      netqty := 0;
      PB-GROUP  iter  <while not eofbit
               and (groupid = rec.itemno)>
          netqty := netqty + rec.qty;
          xread(infile, rec);
      PB-GROUP  end
      writeln(' ', groupid, ' ', n   etqty);
    PB-REPORTBODY  end
    close(infile);
PB  end
```

```
PB'  seq
   PAI
   writeln(' Daily Net Movement Summary ');
   writeln;
   PB-REPORTBODY  iter <while noteofbit>
      groupid := rec.itemno;
      netqty := 0;
      PB-GROUP  iter  <while not eofbit
               and (groupid = rec.itemno)>
          netqty := netqty + rec.qty;
          PAI
      PB-GROUP  end
      writeln(' ', groupid, ' ', n   etqty);
    PB-REPORTBODY  end
PB  end
```

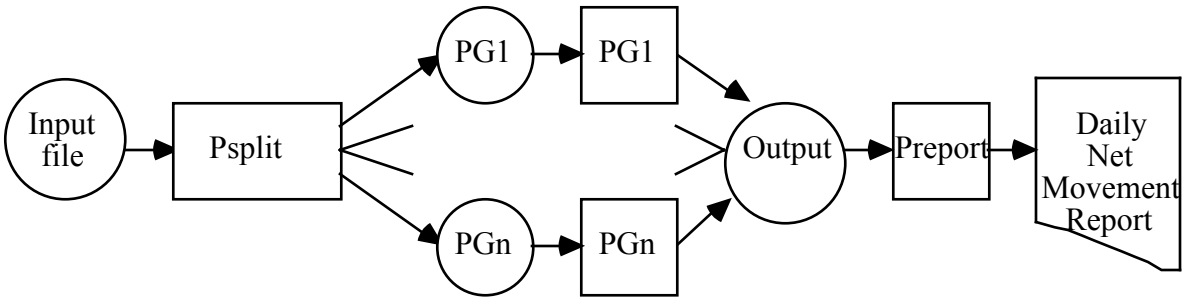
The following changes transform PB to PB':

(1) The initial reset(infile) and xread(infile, rec) in PB are replaced by an initial invocation of the procedure PAI;

(2) The xread statement in PB-GROUPBODY is replaced by invoking PAI;

(3) The close(infile) statement in PB is deleted in PAI.

Recall the following system diagram for the interleaving problem:



We created a multiplicity of files, PG1...PGn. We can rid of these by inverting each of the programs PG1...PGn into procedures PGp_1...PGp_n. We still have a multiplicity of procedures. We observe that the procedures have identical text--they are the same--what differs is only their state vectors. If we separate the procedure text from the state vectors, we obtain just one procedure, PGp, and must simply provide a way for the procedure to access the correct state vector (we can store the state vector for group i as the
i th record in a direct access method, or as the i th row of an array, for example). Separation of the text of a program from its state vector is another implementation transformation in JSP that optimizes design without introducing design errors.

7.3 Implementation of Inversion

In the Pascal implementation of inversion, we use the case construction instead of the construct introduced in our structure text, namely,

GOTO(l_1,l_2,...) DEPENDING ON QS

to express the variable states of PAI. The case statement equivalent is shown below:

case qs **of**
l_1: <statement-1>;
...
l_n: <statement-n>;
end;

The case statement is, of course, a general selection statement, and is thus not very satisfactory since what we have in PAI is not selection, but a strictly determined sequence of states: qs is 0 first, then qs is 1, etc.

Since procedures in Pascal do not remember their state, the state vector of PAI must either be global variables, parameters of PAI, or the entire state vector needs to be saved when PAI is suspended and restored when PAI is resumed. Finally, the state variable, qs, must be initialized at the start.

In order to implement the structure text of PAI in Pascal, we will have to rid ourselves of its nested block structure, since we resume control at a point within an iteration (which is itself contained in another iteration) and Pascal does not allow us to jump into the middle of an iteration. In general, 'nest-free' or flat code must be generated for control structures as shown below:

Structure text	Nest-free Pascal
Sequence	
A **seq**	**begin**
do B;	B;
A **end**	**end**
Selection	
A **sel** <cond-1>	**if not** <cond-1> **then**
do B;	**goto** l_1;
A **alt** <cond-2>	B;
do C;	**goto** l_2;
A **end**	l_1: **if not** <cond-2> **then**
	goto l_2;
	C;
	l_2: {end}
Iteration	
A **iter while** <cond>	l_1: **if not** <cond> **then**
do B;	**goto** l_2;
A **end**	B;
goto l_1;	
	l_2: {end}
Backtracking (selection)	
A **posit**	l_1: {posit}
do B;	B;
A **quit** <cond>	**if** <cond> **then**
do C;	**goto** l_2;
{quit posit, go admit}	
A **admit**	C;
do D;	**goto** l_3;
A **end**	l_2: {admit}
	D;
	l_3: {end}

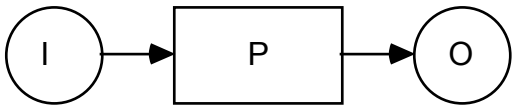
Backtracking (iteration)

```
A  iter                          l1:   {iter}
   do B;                               B;
A  quit <cond>                         if <cond> then
   do C;                                     goto l2;
A  end                                 C;
                                       goto l1;
                                 l2:   {end}
```

Jackson and his co-workers developed specialized compiler pre-processors to generate nest-free code for both COBOL and PL/I. There is no such pre-processor for Pascal, and we will therefore not pursue implementation of inversion in Pascal any further. Also, recent developments in programming languages permit direct modeling of coroutines[9].

7.4 Significance of Program Inversion

We have looked at program inversion as a method for optimizing simple programs connected by serial data streams as in the system diagram below:



This schema of a simple program would seem to have limited applicability to batch data processing systems.

However, the deeper significance of program inversion is that many situations appearing in their dynamic, piecemeal executable form can be recast in their underlying serial form as a simple program. Any resumable program--one that is alternately activated and suspended--is an example of inversion. We can ask 'What is the underlying seriality of its input and output?' Once we discover the underlying seriality of the problem, we can recast the problem in serial form, and design a simple program using JSP. Then, confident of the correctness of our design, we can optimize the design using inversion. Since the inversion preserves program correctness--it is an algorithmic transformation--we can be confident about the design of the inverted (resumable) program. The recognition of resumable processes as being essentially the inversion of a serial process, together with the correctness-preserving property of the inversion transformation of a simple program, allow us to extend the range of JSP to many situations that at first glance do not appear to be amenable to it. Some examples are the

---

[9] C has a number of features that make implementation of inversion easier: for example, static variables, a quit statement, and branching into loop constructs.
Modula-2 and ADA model concurrency. Some OOP languages such as POOL model concurrency. SmallTalk has facilities to model parallel processing and process scheduling.

design of interrupt handlers, conversation dialogues, and on-line transaction monitors among others.

In each of these cases, an ongoing process appears dynamically in its executable form. An interrupt handler is activated and processes an interrupt or a dialogue procedure responds to a terminal input, for example. In each case, we approach the problem by viewing it statically, that is, by taking a 'bird's eye' view of the entire temporal process of which we are seeing a snapshot. We view the entire stream of interrupts or terminal inputs in the course of a day; then, having thus recast the problem into its underlying serial structure, we design a simple program using JSP; and, once the program has been designed, we can invert the program into a resumable procedure, confident that its design is also correct since the inversion transformation preserves correctness.

So, the significance of program inversion is that it extends the applicability of JSP into new areas. Let us examine some of these areas--the design of variable state procedures; networks; and data processing systems.

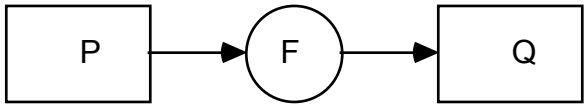
(1) We can use JSP to design variable state procedures.

First we specify the main program equivalent to the procedure; then we design the main program; finally, we invert it.

Here are some examples: [10]

(a) Q requires a procedure P that returns the first prime on the first invocation and the next prime on each of the next 999 invocations.

The equivalent main program, P, creates a file of 1000 prime numbers that is read by the program Q, as shown in the system diagram below:

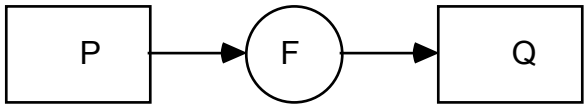


P is inverted with respect to F to create a variable state subroutine invoked by Q.

(b) Q requires a procedure P that scans a database and returns for each invocation, all database customer segments that satisfy condition C.

[10] Examples (a) and (b) are from "Program Inversion and Its Consequences" in JSP&JSD: The Jackson Approach to Software Development" by J. R. Cameron. 1983 (1st edition); example (c), (d) and (e) are adapted from "Constructive Methods of Program Design" by M. A. Jackson in the same book, p. 81.

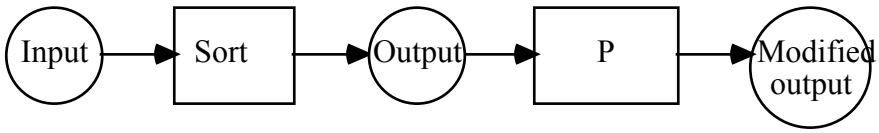
The equivalent main program, P, writes a file, F, of all customer segments satisfying the criteria C, and the program Q processes this file as shown in the system diagram below:



A variable state subroutine can be created by inverting P with respect to F.

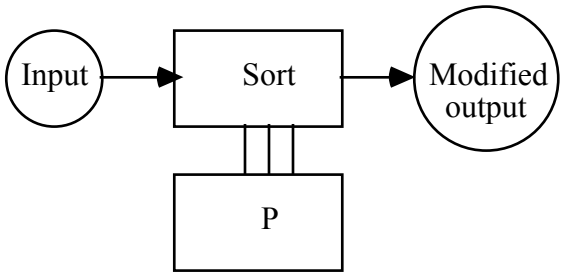
(c) Q requires a procedure P that will modify the output record of a sort program before it is written to the output file.

We may view the system diagram of the sort as below:



The equivalent main program, P, reads the sorted output file and modifies each record, creating a modified output file.

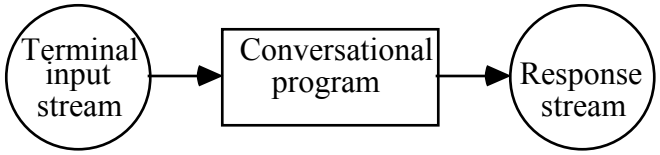
Here, P is inverted with respect to both its input and output files. We say, in this case, that P is doubly inverted and depict the situation with diagram below:



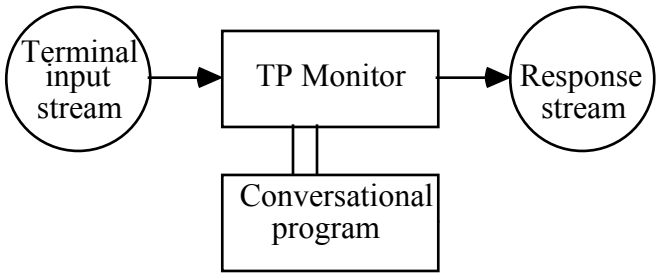
The triple lines (instead of the double lines indicating inversion with respect to one file) indicate inversion with respect to two files.

(d) Interactive conversational programs

To design a procedure to output a response to a terminal input, first design a program to process the entire stream of terminal inputs producing a stream of responses as shown in the system diagram below:



Inverting the resulting program with respect to the terminal input stream gives the desired procedure which we may imagine runs under control of a teleprocessing monitor (TM) as shown below:



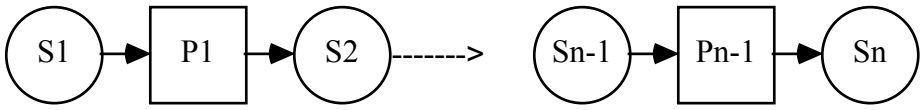
(e) Interrupt handler

To design a procedure to respond to a given interrupt, first design an interrupt handler that reads an input stream of interrupts and outputs a stream of responses; then invert the program with respect to its input stream to give the desired procedure.

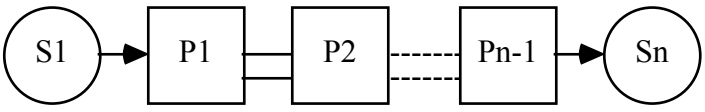
(2) Implementation of a hierarchical network

Problems are sometimes implemented as a network of programs connected by serial data streams. The network can be simplified through inversion as a hierarchy of subroutines.

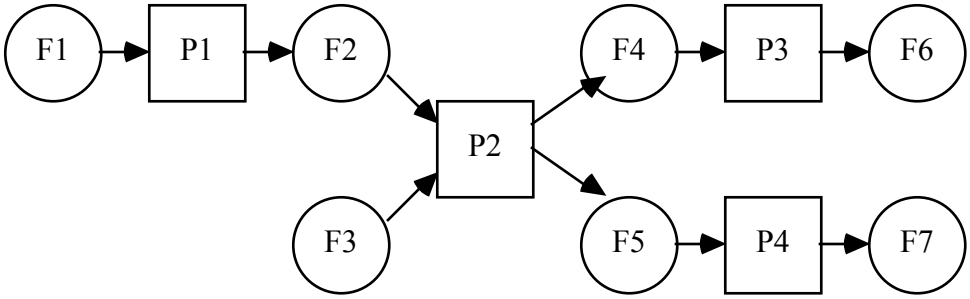
Consider the linear network below (suggestive of 'pipelining'):



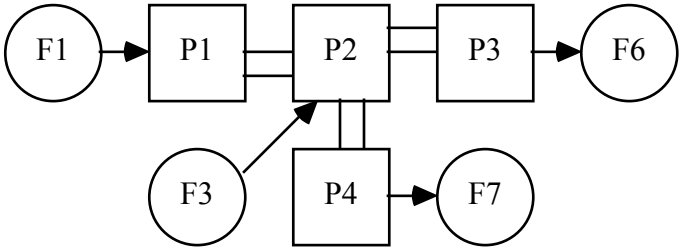
Each of the programs, $P_2,...,P_{n-1}$ can be inverted with respect to its input file, producing the inverted system diagram shown below:



The hierarchical network shown below:

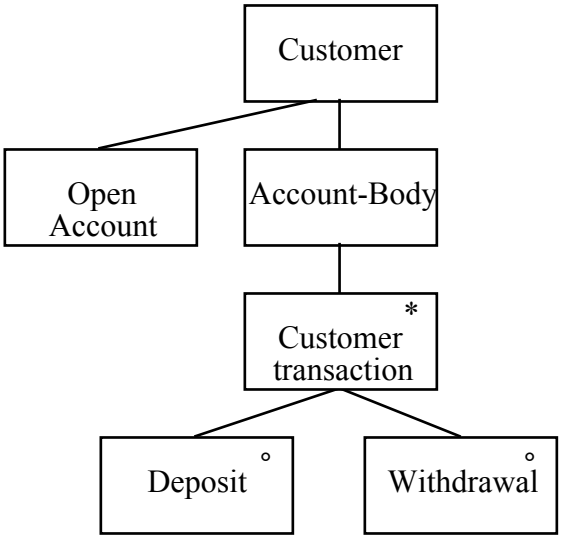


can be inverted, producing:



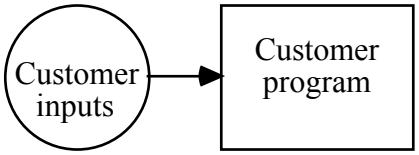
(3) Design of data processing systems

In most data processing systems, entities have a long life--often years or decades. Viewed dynamically, entities--such as a bank customer who makes deposits or withdrawals--are active only briefly; most of the time, the bank customer is engaged in other activities having nothing to do with banking. Let us consider a simple banking system where customers may open an account and subsequently make any number of withdrawals or deposits. The structure diagram for any customer is:

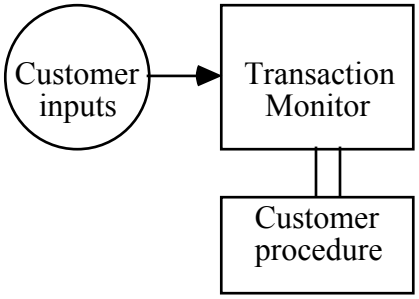


When active, a bank customer's state vector must be activated and the program text for that customer resumed where it left off previously.

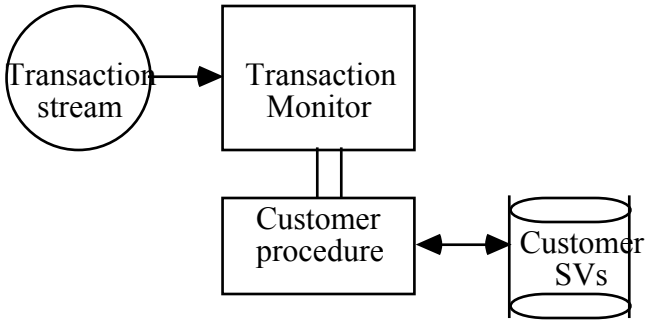
Program inversion allows us to exploit long-running processes to the full. We design the program text of a bank customer based on the structure of its input stream of deposits and withdrawals which may be created over a period of many years as shown below:



Then we invert the program with respect to its input stream, and perhaps run the resulting procedure under control of a transaction processing monitor for example. In the diagram below, a customer process is activated briefly only when necessary; otherwise, it remains suspended.



An information system may be viewed as a network of processes. Each process describes an independent entity in the real world model and is represented by a program. By arranging entities of the same type (e.g., the customers in a banking system that provides customer checking accounts) into sets, we can arrange the corresponding individual programs into an equivalence class: the program text is the same for all entities (customers) in the set; only the state vector--the local variables of the process together with its text pointer--of individual entities (customers) differs. In the diagram below, we show an input stream of transactions for all customers.



The transaction monitor activates the inverted customer program, whose program text is the same for all customers. The program looks up the state vector of the customer-id which is stored on disk (together with the state vectors of all customers) and updates it. The customer program then returns control to the transaction monitor and is suspended.

We will be focusing our attention on the design of information systems in Part 2.

Exercises:

(i) Read a sequence of 80-character records and print the characters on a line printer 125 characters per line. Every input record should have a space appended, and the last line should be filled with blanks if necessary. (from C. A. R. Hoare, "Communicating Sequential Processes", CACM V21 N8)

(ii) Implement the program PB and the (inverted program) procedure PAI in Pascal.

(iii) Implement PB with respect to I, creating the procedure PBI. Give the structure text of PA and PBI and implement them in Pascal.

8. Optimization

8.1 Attitude towards optimization

The word 'optimize' means 'to make the best' (from the Latin *optimum* - 'best'). When we use the term optimization to describe making programs as small and/or as fast as possible, it is ill-chosen because in optimizing we may actually make a program bad in other ways--make it less intelligible, harder to maintain, and more prone to error.

Our attitude towards optimization is summarized in the following two rules:

Rule 1: Don't do it

This rule says that program optimization often isn't necessary. We need a quantitative justification for it. It may not matter whether or not a program uses 10% less memory or runs 5% faster. Optimizing memory isn't important if we have plenty of memory available; a 5% savings in time surely doesn't warrant the time and expense of optimization if we are only running a program a few times.

Rule 2: Don't do it yet

This rule states that if optimization is necessary, then before optimizing, we should begin with an unoptimized design that reflects the problem structure. Only with an unoptimized design can we fully understand the program; and without such an understanding, we are likely to introduce logical errors when we optimize.

8.2 Types of optimization

We can distinguish between three kinds of optimization. We will focus only on the third type, optimization by program transformation, because this type of optimization can distort the original, unoptimized program structure that directly reflects the problem structure.

1. Optimization by tuning

Here we refer to strategies such as the following:

- adjusting a file's block size to effect the best compromise between available space and the number of data transfer operations;
- choosing data types to effect the best compromise between file storage space and program processing time for data conversion;
- adjusting queue sizes to obtain the best compromise between space requirements and device utilization;
- adjusting the number of buffers to obtain the best compromise between memory use and database processing efficiency;
- program segmentation to optimize memory via overlays

With the possible exception of the last example above, none of these optimizations affects program structure.

2. Optimization by algorithm

Sometimes we can improve the performance of an algorithm by refining the algorithm itself. For example, to sort a file of n items into ascending sequence using the 'bubble' sort algorithm, we make n-1 passes of the file. In each pass, we 'bubble' one item as far to the right as possible by exchanging each successive item with its right neighbor if it compares greater. For example, if our n items are the integers below:

10 4 5 6 3 8

then, the arrangements below show the result of exchanging after each pass:

n = 6	pass
4 5 6 3 8 10	1
4 5 3 6 8 10	2
4 3 5 6 8 10	3
3 4 5 6 8 10	4
3 4 5 6 8 10	5

We can optimize the algorithm in several ways:

- stop the sort if during a pass, no exchanges were made, instead of making (n-1) passes; the 5th pass above would be unnecessary;
- reduce the number of items involved during each pass: if in the nth pass, the leftmost pair swapped was s_n and s_{n+1}, then the (n+1) pass may begin with the s_{n-1} and sn item; in the 3rd pass, we could start by comparing the 2nd and 3rd items, since in the previous pass, the 1st exchange occurred between the 3rd and 4th items

Other optimization of the 'bubble' sort are possible; in each case, the changes in program structure reflect a new, optimized algorithm.

3. Optimization by program transformation

Here we are concerned with optimizations that could be performed by a good optimizing compiler on a well-designed program. They include eliminating redundant assignment operations; optimizing common subexpressions; loop optimization; etc.

If an optimizing compiler is used to effect program optimizations such as those mentioned above, the simplicity and clarity of the original, unoptimized program remains the basis of future maintenance activity. There is no cost to the optimization except the extra time that an optimizing compiler may take.

Often, however, optimization by program transformation is carried out by the designer, and we run the risk of increasing program maintenance costs through loss of the

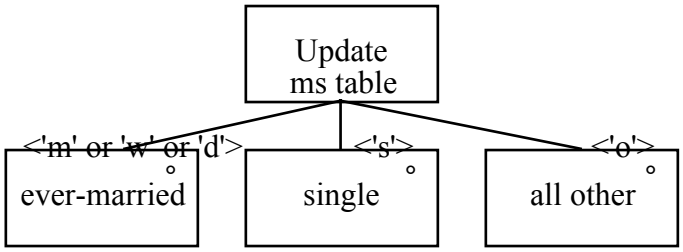
initial clarity and simplicity of design. Let us consider a few such optimizing program transformations.

(a) Ordering of condition tests

Sometimes, we can achieve reduction in execution speed by carefully ordering condition tests in a selection as in the following example:

> A person's marital status in the person record of a population Census file has the following codes and estimated frequencies:
> divorced - 'd' (4%), widowed - 'w' (10%), married - 'm' (30%), single - 's' (55%), other - 'o' (1%)
> We wish to compute the frequency distribution by marital status for the categories single, ever-married (married+divorced+widowed) and other.

The structure diagram for the selection is shown below:



In a structure diagram, recall that the ordering of the selection is unspecified. We might code the selection as follows:

```
type
    maritalstatus = ('d', 'w', 'm', 's', 'o');
var
    ms: maritalstatus;
    ethtab:  array[1..3] of integer;

    case  ms of
's':  ethtab[1] := ethtab[1] + 1;
'm', 'w', 'd':  ethtab[2] := ethtab[2] + 1;
'o':  ethtab[3] := ethtab[3] + 1;
    end
```

Without knowing the details of how the case statement is implemented, we cannot be certain that its use as shown above will optimize the selection processing.

The selection can be ordered in order of decreasing frequency (with the greatest frequency first) to minimize the number of condition tests performed as shown below:

```
/*  selection is optimized to minimize number of condition tests made based on */
/*  estimated frequencies of marital status codes */
if ms = 's' then
     ethtab[1] := ethtab[1] + 1
     else if ms = 'm'  then
          ethtab[2] := ethtab[2] + 1
          else if ms = 'w' then
               ethtab[2] := ethtab[2] + 1
               else if  ms = 'd' then
                    ethtab[2] := ethtab[2] + 1
                    else ethtab[3] := ethtab[3] + 1;
```

There is some loss in intelligibility, unless comments are included to indicate that the selection has been optimized as shown.

The remaining three kinds of program transformation considered below optimize by reducing program length:

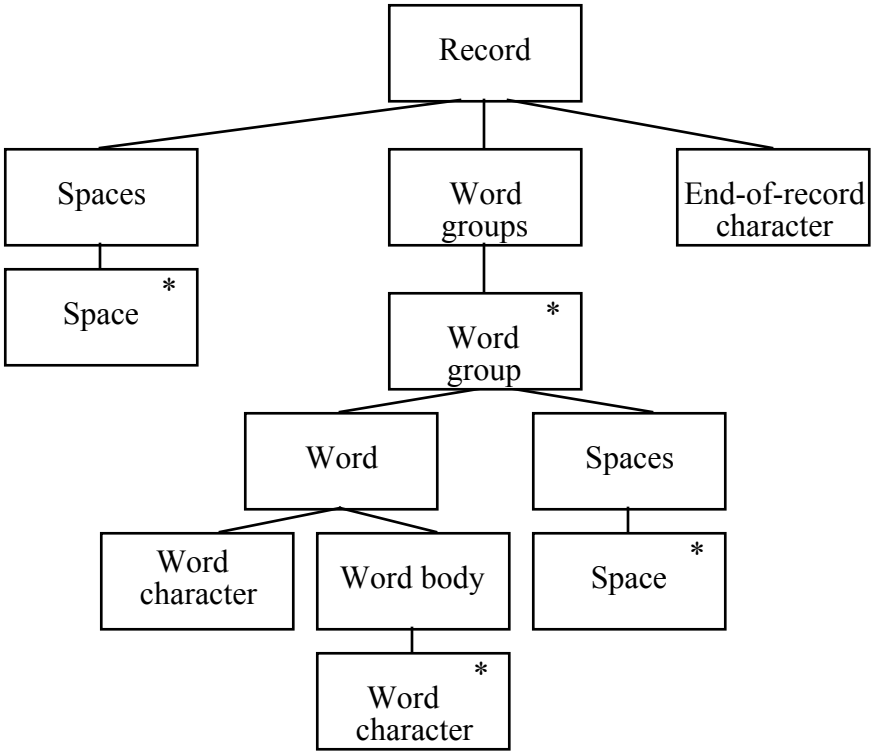
(b) Simplification of data structure

Simplification of the data structures on which the program structure is based must be done with care since an error or unwise optimization could undermine the entire program structure.

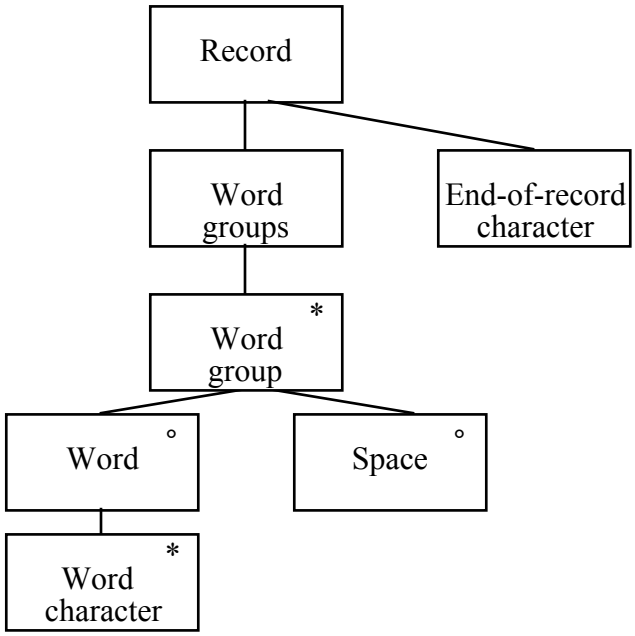
Consider the following description of a record:

> A variable-length record (maximum length = 1000) begins with some number of spaces followed by words, each of which is separated by one or more spaces, and is terminated by a distinct end-of-record code. Each word has at least one character and consists only of the upper and lower case letters a-z, and single and double quotes.

The record structure is depicted below:



We can, in fact, simplify the data structure considerably, thereby reducing the number of components and the program's length:



We have eliminated the initial iteration of spaces; the iteration of spaces following a word; and a word's first character. We can confidently process records, as

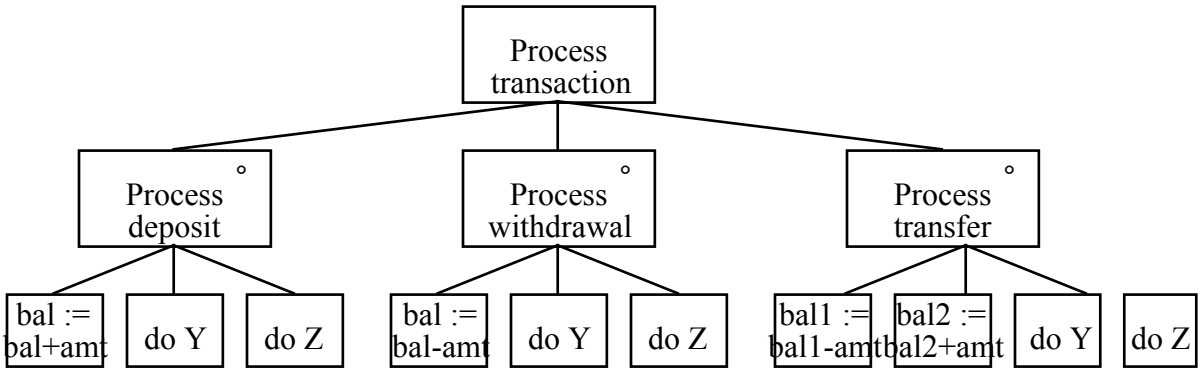
long as in the specification, the delimiters between words are limited to spaces and the number of spaces at the beginning and between words is unimportant.

(c) Common action tails

Consider the following problem:

> A bank customer may make a deposit , withdrawal or transfer. In the case of a deposit, the amount is added to the account's balance and then a record of the account activity is recorded by executing procedures y and z; in the case of a withdrawal, the amount is subtracted from the customer balance, and a record of the account activity is recorded by executing procedures y and z; in the case of a transfer, the transaction amount is subtracted from the first account number and added to the second account number on the transaction; following this, a record of account activity is recorded by executing procedures y and z.

Consider the following structure diagram:



We notice that for each transaction type, we execute procedures Y and Z subsequent to updating the customer accounts involved. We could optimize as shown in the following structure text:

```
PROCESS-TRANS    sel
    bal := bal + amt;
1:  do Y;
    do Z;
PROCESS-TRANS    alt
    bal := bal - amt;
    goto 1; /* common action tail */
PROCESS-TRANS    alt
    bal1 := bal1 - amt;
    bal2 := bal2 + amt;
    goto 1;    /*  common action tail  */
PROCESS-TRANS    end
```

The use of the goto statement is deliberate--the original selection structure is preserved. It would be less intelligible to restructure the selection based on the optimization as shown below:

```
PROCESS-TRANS   seq
    PROCESS-TRANS-SEL  sel
        bal := bal + amt;
    PROCESS-TRANS-SEL  alt
        bal := bal - amt;
    PROCESS-TRANS-SEL  alt
        bal1 := bal1 - amt;
        bal2 := bal2 + amt;
    PROCESS-TRANS-SEL  end
    do x;
    do y;
PROCESS-TRANS   end
```

If a change is made in the processing of any of the three transaction types, we would likely have to rewrite the component.

(d) Generalization

Another way to handle two or more identical lines of code that occur in more than one place in a program is to generalize by making the identical lines of code into a procedure. Thus, in the common action tail example above, we could make the procedure shown below:

```
    procedure otherprocessing;
    begin
        do x;
        do y;
    end;
```

The component to process the transactions is now:

```
PROCESS-TRANS    sel
    bal := bal + amt;
    do otherprocessing;
PROCESS-TRANS    alt
    bal := bal - amt;
    do otherprocessing;
PROCESS-TRANS    alt
    bal1 := bal1 - amt;
    bal2 := bal2 + amt;
```

```
        do otherprocessing;
    PROCESS-TRANS    end
```

Optimization by generalization should be sharply contrasted with the generalization that occurs in bottom-up design, where we build a more abstract machine in order to include operations needed by the problem environment (see section 2.4).

9. Concluding remarks on JSP:

9.1 JSP and the Design of Programming Languages

We have seen a number of cases where we have had to invent structure text constructs to implement the design methods of JSP because the control structures were not available in Pascal. JSP suggests that programming languages should include features that directly implement the JSP design method.

Let us review these features briefly:

(1) We need a **posit** and **admit** control structure and a **quit** statement to express backtracking

Although we used a **goto** statement to implement backtracking, this was not an unrestricted **goto** statement of the type that Dijkstra and others have warned against; it is a carefully limited **goto** that is semantically equivalent to the **admit** structure text.

(2) We need a coroutine facility that permits one process to be suspended while control is passed to the appropriate point of the invoking process, and that permits the suspended process to be invoked again and resumed immediately following the point at which it last relinquished control.

Since such a coroutine facility is not available in many languages, including Pascal, we have used a variable state procedure to model the suspend-and -resume mechanism. As was mentioned in section 7.3, for Pascal just as for COBOL and PL/1, we need a special-purpose compiler that generates nest-free Pascal code in order to resume at a point within a nested block, such as an iteration. Although the implementation of nest-free code generators is part of JSP implementation, we will not discuss nest-free code further beyond reiterating the following points:

- Special compilers can be written to produce nest-free code to permit implementation of variable state subroutines;

- The **goto** statements generated in nest-free code are not 'harmful' since they are carefully controlled and semantically equivalent to the nested block structure that they implement in unnested form.

- The essence of the 'suspend-and-resume' mechanism is that a procedure remember its state vector (local variables and text pointer). Objects in OOP languages do remember their state, or at least their local variables. OOP languages that support concurrent processes directly remember their text pointer as well; in other OOP languages, the text pointer must be modeled by a local variable, as described in the chapter on program inversion.

9.2 Simple programs and serial data streams

JSP provides a method for designing simple programs. A simple program, shown below, has the following characteristics:

(a) a fixed initial state
(b) program inputs and outputs are serial files
(c) input and output files have an explicit structure
(d) input structures define the domain of a program; output structures define the range of a program
(e) input and output data structures are compatible in the sense that they can be combined into a single program structure

We have seen that more complex problems can often be decomposed into simple programs, which can be designed using JSP and optimized by inverting one or more programs with respect to its (their) data file(s).

Jackson proposes that the **simple program**--rather than the module of modular programming or the control structure of structured programming--provides a reliable, high-level design component for software engineering. In particular, serial data streams form a simple and effective interface--a barrier that enforces a design discipline--between programs.

By a serial data stream we do not mean a sequential data set in which 'next' is based on the value of a record identifier. Instead, the idea of seriality is that of a physical seriality of data produced by some process.

Examples of serial data streams include:

- a stream of messages input from a terminal (they reflect the temporal sequence of a user's responses in a conversational dialogue)
- the sequence of records in a physically serial file (some process created the physical ordering of the file)
- a stream of data base segments accessed in a sequence of database calls (the stream reflects the process of accessing a specified set of data)
- a sequence of invocations of a procedure (the sequence reflects the process of executing the invoking program)
- a sequence of characters printed out in a memory dump (the sequence reflects the program's access of memory locations)

When we come to view system development (JSD) in the next part of the text, we will see that a system can be specified as a network of entity model processes, with each

model process described (and implemented) by a simple program that reads a serial data stream connecting it with the real world.

Thus, the idea of a simple program in JSP is very close to the idea in the object-oriented design paradigm of objects communicating with each other by passing messages. Over time, the sequence of messages read by an object constitutes precisely a serial data stream.

10. Jackson System Development (JSD): An Overview

10.1 A Simple Example: Student Loan System

Program inversion extends the range of JSP applications into various areas such as coroutine design, implementing a system of simple programs as a hierarchy of subroutines, on-line programming and systems design. We will now look at JSD, a method for design information systems.

Consider the following description of a student loan system:[11]

> A university gives loans to students. Before getting a loan, there is an evaluation process after which agreement is always reached. A TE transaction records each step of the evaluation process, and a TA transaction records the overall loan agreement. A student can take any number of loans, but only one can be active at any time. Each loan is initiated by a TI transaction. Then, the student repays the loan with a series of repayments. Each repayment transaction is recorded by a TR transaction. Finally, a loan is terminated by a TT transaction.
>
> Two output functions are desired: (1) an inquiry function that prints out the loan balance for any student, and (2) a repayment acknowledgment sent to each student after payment is received by the university.

The university loan office decides to implement the student loans on a single processor. Inquiries should be processed as soon as they are received. However, repayment acknowledgments need only be processed at the end of each day.

Notice that each student who receives one or more loans generates a stream of data over a long-period of time, perhaps many years. We can write a program to process the whole data stream, if we can describe its structure, just as we can write a program to process the data input by a user at a terminal. A key feature of many information systems is that they consist of slow, long-running processes. JSD takes a bird's eye view --a holistic view-- of such processes, describing the entire serial file even though it has not yet been produced.

10.2 Modeling Phase

JSD consists of three main phases: the modeling phase; the network phase; and the implementation phase. In this section, we will look at the modeling phase.

The modeling phase consists of three steps:

Step 1: Entity/action step

---

[11] adapted from [Ja75], Ch. 11

The first step in modeling a real-world system, is to decide on what entities are relevant in the system. An entity is any object--a person or thing--that is important in the system we are modeling. Looking at the actions that occur in the system helps us to decide what entities are important. An action is always associated with an entity.

Actions have the following characteristics:

(1) An action takes place at a point in time

The evaluation process, taken as a whole, is not an action, since it takes place over a period of time and consists of individual subactions.

(2) An action must take place in the real world outside of the system.

The system action of issuing a 'loan repayment acknowledgment' is not an action, since it is an action of the system. A loan repayment by the student is signified by a 'loan repayment action', so one cannot suppose that a 'loan repayment acknowledgment' somehow models a student's loan repayment in the real world.

(3) An action is atomic, cannot be divided into subactions.

The evaluation process, taken as a whole, is a composite action consisting of some number of evaluation subactions. Each 'evaluate' subaction may be an action (of bank or of student).

Entities have the following characteristics:

(1) An entity performs or suffers actions in time.

A statistic, such as population of U.S. in 1980 isn't an entity, nor is U.S. or the year 1980 or any other object in a static database; A bank that issues loans over time or a student who makes loan repayments over time are objects that perform time-ordered actions.

(2) An entity must exist in the real world, and not be a construct of a system that models the real world

Thus, a 'Loan repayment acknowledgment' issued by the system is not an entity.

(3) An entity must be capable of being regarded as an individual; and, if there are many entities of the same type, of being uniquely named.

A student is a possible entity. There are many students (who belong to the entity type or class, student), but each individual can be named.

We begin by making a list of candidate entities and actions. For each candidate action, we decide what entity the action is associated with, and provide a list of attributes associated with the action.

Here are the candidate lists:

Entities/Description:

student
system
university
loan
student-loan

Actions/Attributes:

evaluate -action of university? (university performs the evaluation);
action
of student? (student is evaluated) stud
attributes: student-id, loan-no, date of evaluation, remarks
agree - action of university? (university agrees to loan); action of student
? (agrees to loan)
attributes: student-id, loan-no, date of agreement, amount of loan,
interest rate, repayment period)
make loan - action of university
attributes: student-id, loan-no, date of loan, loan amount, interest rate,
repayment period
initiate - action of university? (university initiates loan); action of student?

(student initiates loan); action of loan? (is initiated)
attributes: student-id, date initiated
repay - action of loan? (loan is repaid);
action of student? (student repays the loan);
attributes: student-id, date of repayment, amount of repayment
terminate - action of loan (loan is terminated);
attributes: student-id, date of termination, remarks

In examining the actions, "make loan" seems vague with respect to its time. Is a loan made when it is agreed to or when it is goes in effect? Probably the former and effective date would be part of the agreement. Two actions, agree and initiate, are specifically mentioned in the description, and "make loan" seems to encompass both, so we eliminate "make loan" from our action list in favor of "agree" and "initiate".

We eliminate "university" from the candidate list of entities, because the system clearly is not about (is not interested in) the university, how it is administered, who the

loan officers might be (except, perhaps, in relation to who authorized a loan to be made), etc. We may say that "university" is outside the model boundary or OMB.

Student, student-loan, and loan give us pause. Students are evaluated before they receive a loan, and the system should model (record) the evaluation process. Hence evaluation occurs before there is any student-loan. We decide to eliminate student-loan in favor of student, where student has the meaning in the loan system of students receiving one or more loans from the university. Loan is well-defined and could be considered, with student, an entity. What actions does a loan perform or suffer over time? Loans are agreed to, initiated, repaid and terminated. What actions do students perform or suffer? Students are evaluated, agree to a loan, initiate the loan, repay the loan and terminate the loan. The structure of a loan is included in the structure of student, and there is no need at present to include loan as a second entity, although this may be considered in later. We reject loan as an entity.

After considerable discussion of these and other points with the university loan office, we decide finally that we have only one entity of interest: student with actions evaluate, agree, initiate, repay and terminate. Our final entity and action lists are as follows:

Entities/Description:

student

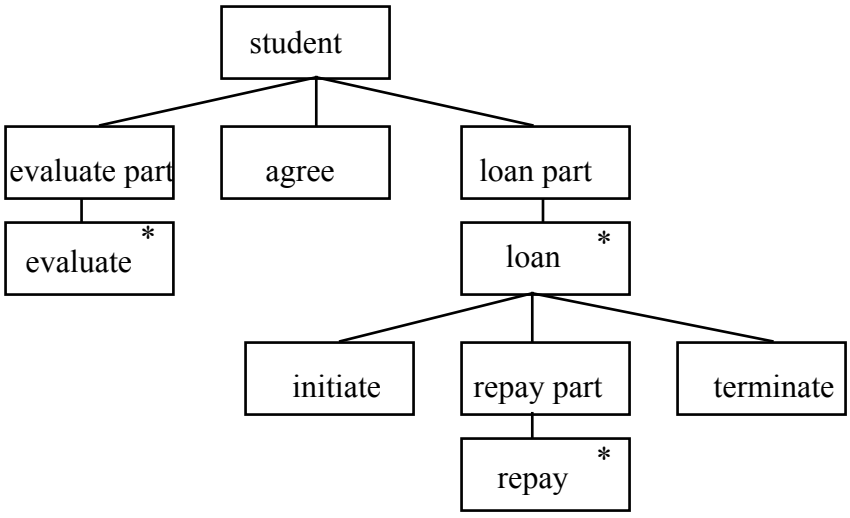
Actions/Attributes:

evaluate -action of student; student? (st
  attributes: student-id, loan-no, date of evaluation, remarks
agree - action of student
  attributes: student-id, loan-no, date of agreement, amount of loan,
interest rate, repayment period)
initiate - action of student
  attributes: student-id, date initiated
repay - action of student
  attributes: student-id, date of repayment, amount of repayment
terminate - action of student
  attributes: student-id, date of termination, remarks

Step 2: Entity structure step

The behavior of an entity in the real-world is modeled in terms of the actions it performs or suffers over time. The second step of JSD expresses the constraints in the ordering of an entity's actions with a structure diagram.

The entity structure diagram for a student is shown below:



The exact meaning of this diagram is:

* The first action suffered by a student is the evaluation part, which consists of zero or more evaluate actions.
* The next action of student is agree.
* Next comes the loan part, which consists of zero or more loans.
* Each loan is a sequence of initiate action, followed by an iteration of repay actions, followed by a terminate action.

Notice that the leaves of the tree are the atomic actions of the entity (analogous to elementary actions in a program structure).

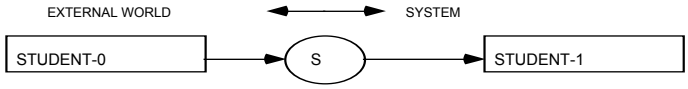
Step 3: Model process step

In the last step of the modeling phase we create an executable **model process** for each entity in the system. Each model process is described with a system diagram.

Our information system models each real-world entity. We are not just simulating entities artificially, using some statistical techniques for example; we are modeling each entity precisely as it exists in the real world. Therefore, there must be a connection between the entity in the real world and the entity model process in our information system.

Usually, the connection is by serial data stream, in which the real world entity produces a message for each action performed (or suffered). For each student action in the real world, a transaction records the action in the student loan information system. Viewed over time, the set of messages recording student actions constitutes a serial data stream.

We can depict the serial data stream connection between student entity and the student in the information system as shown below:



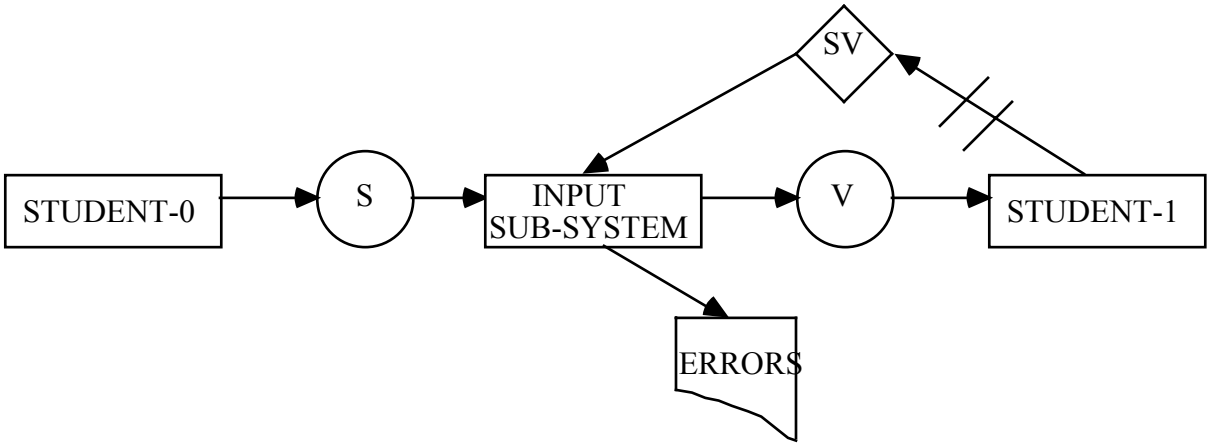
We use the suffix "-0" to the student entity to indicate the abstract student entity in the real world, and the suffix "-1" to indicate the realization of this abstraction in our information system.

What is the structure of the serial data stream? The entity structure diagram describes its structure. The entity model process for a student can be described by a simple program that reads a serial data stream. We can use JSP to write the program text for this process. It is shown below:

```
STUDENT-1 seq
    read S;
    EVAL   iter (while TE)
        process TE; read S
    EVAL  end
    AGREE  seq
        process TA; read S
    AGREE  end
    LOAN-PART iter (forever)
        INIT   seq
          process TI; read S
        INIT
        REPAY  iter (while TR)
            process TR; read S
        REPAY  end
        TERM   seq
          process TT; read S
        TERM  end
    LOAN-PART end
STUDENT-1 end
```

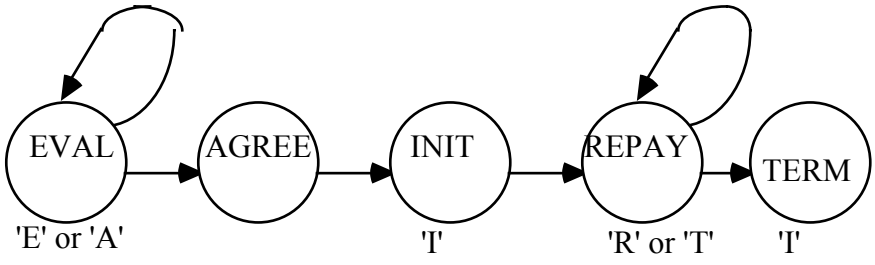
Although it is a slow-running system, our information system is a real-time system. We collect information as it is furnished by the real-word process. Our entity model process is synchronized with the actions of the real world entity. Recall that the text pointer is part of the state vector of a model process's program text. If our text pointer points to the repay component of a student's process text, then an 'E' (evaluate), 'A' (agree) or 'I' (initiate) transaction on the input stream for this student is clearly unacceptable and must be recognized as an error.

Error handling is defined in this step to filter out errors so that the realized model simulates the real world faithfully. We may view the input system as being interposed between the real world entity and our system, as shown in the SSD below:



The function of the input subsystem is to ensure that only valid transactions, indicated by the data stream V, are input to the specification of STUDENT-1. Let us suppose that in our student loan system, we choose to simply reject those transactions that are not valid for the given state of a particular student's model process. We can examine the current state of the student model process by inspecting its state vector. We will use a variable, **state**, to keep track of a student's current state, initializing it to 'eval' when a student model process is first instantiated. We will assume that **state** and the rest of a student's state vector reflect the point at which read V operations occur in STUDENT-1's text as follows: Either an 'E' or an 'A' transaction (evaluation or agree action) is acceptable for a new student model process initially. After an agree transaction has been processed, **state** must be set to 'init'. The only acceptable transaction is an 'I' (initiate loan). After processing the initiate loan transaction, **state** is set to 'repay'. Acceptable transactions are either 'R' or 'T' (repay or terminate). After a 'T' transaction has been processed, **state** is set to 'init', indicating that the student process is expecting to initiate another loan following termination of the last one.

Referring to STUDENT-1's process text, we can compose the following state transition diagram containing labeled states corresponding to the components of the STUDENT-1 process (EVAL, AGREE, INIT, REPAY and TERM), and the acceptable transactions for each state.

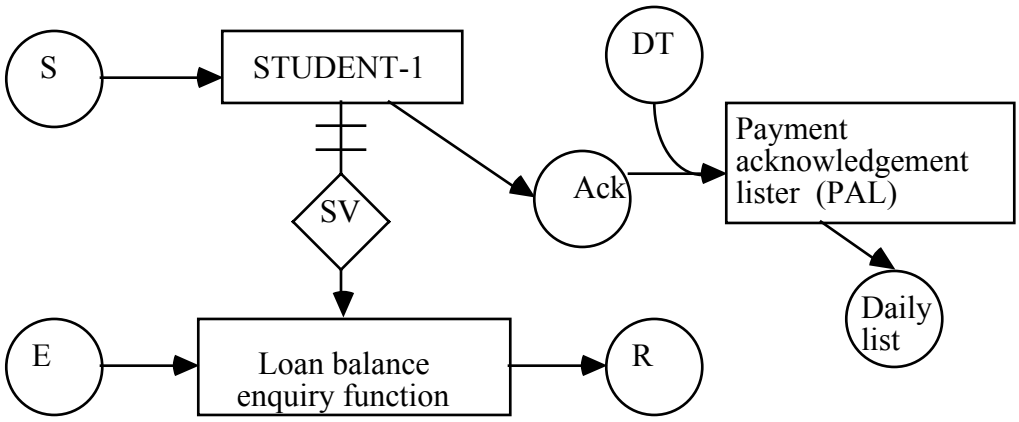


The program text for the input sub-system is given below:

```
INPUT-SUBSYSTEM  seq
    read S(transrec);
    FILTER iter  <while not eof>
        getsv(student-id);
        ACCEPT-TRANS   sel  <trans-code valid for student-id current state>
            write V(transrec);
        ACCEPT-TRANS  alt
        ACCEPT-TRANS  end
        read S(transrec);
    FILTER  end
INPUT-SUBSYSTEM  end
```

10.3 Network Phase

In the network phase, we connect model processes and functions in a single system specification diagram (SSD). In our system, we have one entity and two functions. The SSD is shown below:



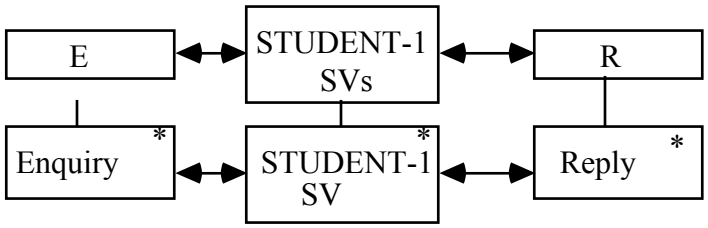
The loan balance inquiry function (LBE) is connected to the Student-1 process by state vector (SV) connection. In this type of connection, one process can examine the state vector of a second process. The double lines indicate that an inquiry process, over its life, will examine many student processes.

The function to produce the student acknowledgments data stream (ACK) is embedded in the student-1 process in the repays component as shown below:

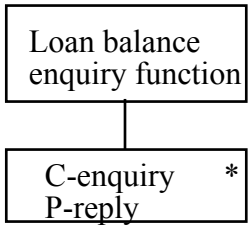
DT is an input signal at the end of the day--a daily time marker--that tells the payment acknowledgment lister (PAL) function to begin. The ACK and DT data streams are rough-merged, that is, we don't know precisely whether a repayment acknowledgment will appear on today's or tomorrow's daily list.

The loan balance inquiry function (LBE) is designed using JSP as shown below:

(i) input and output data structures:

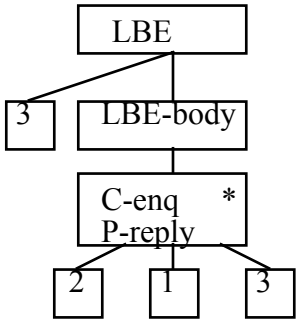


(ii) basic program structure



(iii) list of operations:
1 - write 'loan balance for', student-id, 'is', balance
2 - get STUDENT SV (student-id)
3 - read E

(iv) elaborated program structure and text:



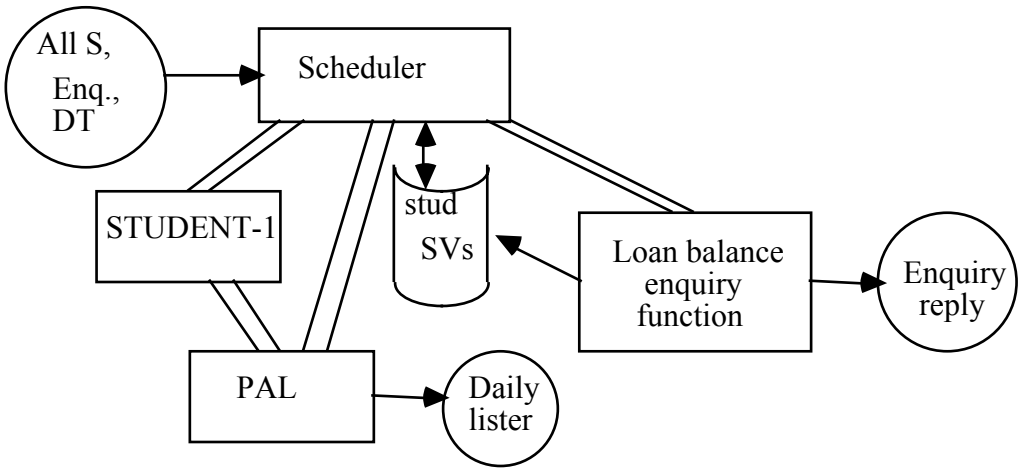

```
LBE     seq
   read E;
   LBE-BODY itr      (forever)
      get STUDENTSV(student-id);
      write 'loan balance for ', student-id,
         ' is ', balance'
      read E;
   LBE-BODY    end
LBE  end
```

10.4 Implementation phase

First we examine the timing constraints of the system. The problem description for our system states that inquiries are to be answered on-line but repayment acknowledgments at the end of the day.

We then consider possible hardware and software for implementing our system. We decide to implement all student processes on a single processor. Each student process could have been allocated a dedicated processor.

Based on hardware and timing constraints, we design a system implementation diagram (SID) shown below:



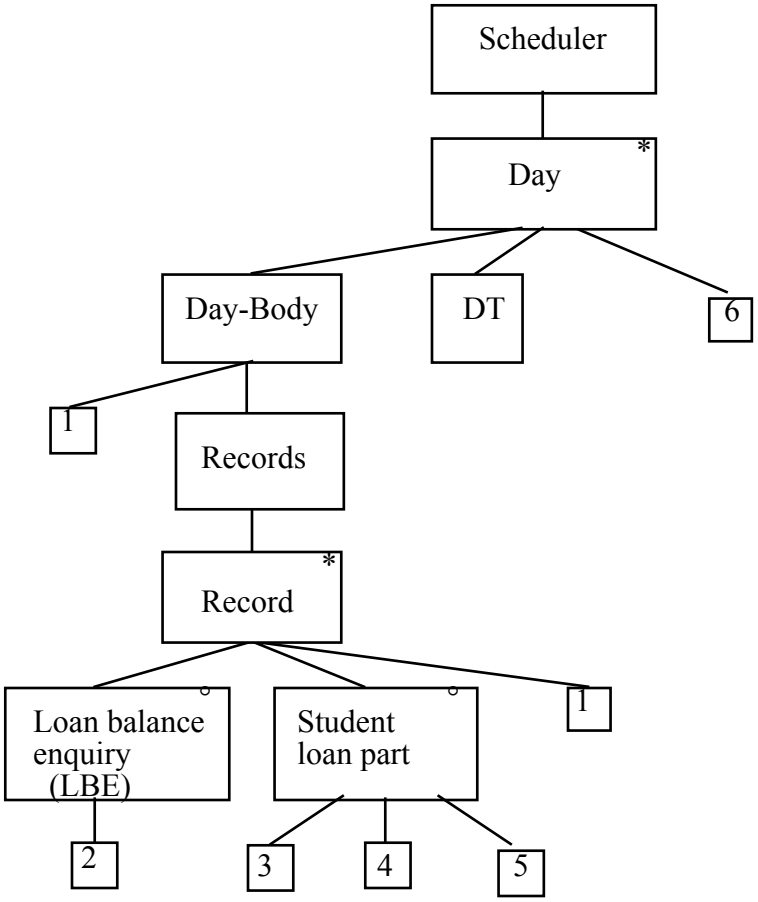
In the SID, all of the serial data streams are input to the scheduler process. Note that the all student processes have an identical structure text; only their state vectors are different. Consequently, we can separate the state vectors of student processes from their process text--this separation is called state vector separation. The set of state vectors constitutes the data base of our student loan system.

The student-1 process is inverted with respect to its data stream, S. Student-1 is called by the scheduler to process a transaction, and then suspended when it has completed the processing of the transaction, with control returning to the scheduler. Note that the scheduler must access the student's state vector corresponding to the student-id on the input data stream, and for saving the updated state vector on completion of processing the transaction.

The loan balance inquiry is invoked by the scheduler when it has read an inquiry.

The repayment acknowledgment lister (PAL) is inverted with respect to both of its inputs, the repayment acknowledgment data stream and the daily marker. PAL is invoked by Student-1 whenever Student-1 processes a repayment transaction. The scheduler invokes PAL directly when it receives a DT and this triggers the daily listing.

The scheduler process is designed with JSP, and is shown below:



List of operations:

1-read input
2-call LBE(inrec)
3-get SSV(student-id)
4-call student-1(srec, ssv)
5-put SSV(student-id)
6-call PAL(DT)

The exact meaning of the scheduler structure diagram is as follows: On any given day, records from the serial data stream (loan balance inquiries and student loan transactions) are read and processed in real-time. At the end of the day, a daily time marker--perhaps a signal to the system from the operator--is input, and the payment acknowledgment lister program is invoked. It processes payment acknowledgments that have been previously generated in real-time whenever a student repayment is made and stored in a buffer.